Guia Front-End

O caminho das pedras para ser um dev Front-End

</>

DIEGO EIS

Sumário

Introdução

Sempre ouvi muitas reclamações de iniciantes dizendo que faltam informações mostrando o "caminho das pedras", ensinando como e por onde começar no mercado de front-end. São tantas tecnologias, metodologias, boas práticas e outros tantos tópicos que surgem todos os dias, que eu confesso: realmente, fica muito difícil acompanhar tudo. Além da sobrecarga enorme que é ficar testando todas as novidades. Eu mesmo tenho que filtrar bastante os assuntos a aprender. Geralmente, eu não me atenho a nada que eu não precise utilizar em algum projeto. Não vou aprender a utilizar um **framework** ou uma ferramenta, simplesmente porque todo mundo está comentando por aí. Eu preciso trabalhar, afinal de contas. Tenho que garantir o Jack Daniels das crianças.

Mas, seria pedir muito ter alguém ao meu lado no início da minha carreira dizendo o que é perda de tempo? Muitos devs que decidem se aventurar aprendem toda a matéria sozinhos. Eu nunca entendi isso. Eu mesmo, quando adolescente inconsequente, não queria, nem a pau, estudar por conta própria. Isso mudou depois que comecei a trabalhar com web. Coisas estranhas da vida! Aprender algo sozinho é interessante! Certamente, você pode acabar perdendo bastante tempo tentando descobrir o que vai torná-lo ou não produtivo. É um processo chato e trabalhoso e nem sempre você acerta. Quando estamos sozinhos, é quase certo que deixaremos passar informações importantes no processo de aprendizado. Já vi tantos bons desenvolvedores front-end que iniciaram sua carreira como autodidatas, mas não tinham ideia do que era controle de versão, WAI-ARIA ou, até mesmo, propriedades corriqueiras do CSS. Não porque eles eram desatentos, mas porque estavam focados em outros tópicos. Isso é normal acontecer quando aprendemos algo de forma não linear, e misturando fontes do conteúdo. Por isso, é tão importante ter alguém que mostre, pelo menos, a direção correta. Você economiza tempo e, talvez, até algum dinheiro nesse processo.

A missão deste guia

Este guia tem a missão de ajudar qualquer pessoa que queira iniciar na área de web. Foquei-me em desenvolvedores front-end porque é a área no qual os profissionais possuem maior familiaridade quando desejam entrar no mercado de web. Mesmo assim, se você é um apenas um curioso sobre o assunto, mas quer entender melhor sobre o tema, talvez este livro possa ajudar. Este guia vai apresentar os assuntos mais básicos e importantes, tentando auxiliar aqueles que já adentraram ou querem adentrar, agora, o mercado de web.

Organização por prioridade

Não separei os tópicos por ordem de dificuldade, mas sim por ordem de **prioridade**. Quero deixar claro que a escolha das prioridades atende a minha opinião. Ou seja, há milhares de pessoas aí afora que pensam diferente de mim. Eu decidi, aqui, o que seria mais ou menos importante aprender para ingressar na área. Outra ressalva que faço é que, talvez, você precise aprender um assunto mais complicado antes de passar para tópicos tido como mais fáceis. Por exemplo: é mais significativo que um dev front-end saiba primeiro o básico de JavaScript do que de SEO. Você até encontra um bom emprego sabendo apenas HTML, CSS e SEO, mas, na maioria dos casos, JavaScript costuma ser mais relevante.

O que este guia não é?

Definitivamente, este guia **não** é um livro didático, logo, ele não vai ensinar nada do começo ao fim, nem tão pouco código. Este livro se propõe a mostrar o "caminho das pedras", ditando o que você precisa aprender, mostrando todos os assuntos pertinentes para se tornar um dev front-end e quais os assuntos mais comentados por aí.

Isso não quer dizer que você precise aprender tudo o que está listado aqui. Pelo contrário! Este livro vai ajudá-lo a decidir o que aprender primeiro. Mas, sem dúvida, você precisa saber que estes assuntos existem. Vou tentar indicar links de referências (em inglês e em português) para estudo durante o percorrer do livro. Tome tempo para visitá-los e estudá-los também. São links com muito conteúdo importante.

Ensinar o "caminho das pedras". Inspirar sua curiosidade. Fazê-lo entrar da maneira correta no mundo do desenvolvimento web. É isso que este livro se propõe a fazer. Boa leitura!

HTML

2.1 Tudo começa e termina no HTML

Se você tivesse que aprender apenas um assunto sobre tudo o que este livro comenta, esse assunto seria, sem dúvida, HTML. Sem o HTML — e peço licença para incomuns comparações —, o CSS não passa de uma periguete emperequetada e o JavaScript, um cara solitário chorando desconsoladamente. O HTML é o ponto focal de tudo o que se faz em desenvolvimento web. Não importa o que você faça, o HTML é o que há de mais importante na web.

Não há como você se tornar um dev front-end sem dominar HTML. Ele estará presente desde o início do projeto e será responsável por muitas partes cruciais durante toda a sua elaboração. O HTML é o código que perpassa todas as equipes e você, sendo um dev front-end, é o responsável por ele. Mesmo que os programadores back-end façam algo errado, é você que zelará pela qualidade, performance, estrutura e semântica do HTML. Se a semântica está mal feita, o site não irá aparecer nos resultados de busca, a acessibilidade ficará prejudicada, a manutenção será danificada com o tempo, não existirá portabilidade e, certamente, você terá problemas gigantes com o código legado. Parece exagero, mas não é. Tudo começa e termina no HTML.

A essência

Eu sei que você já deve ter lido exaustivamente sobre a sintaxe do HTML. Mas eu próprio li, em poucos lugares, a verdadeira importância do HTML. Por esse motivo, escolhi dois assuntos interessantes que lhe darão alguma vantagem sobre o idiota, sentado ao seu lado, no dia da entrevista de emprego. Listo: hipertexto e marcação (estruturação semântica). Você pode aprender a escrever código HTML em qualquer lugar na web. Existem milhares de sites brasileiros e estrangeiros que ensinam isso. Contudo, material sobre hipertexto ou semântica são geralmente escassos e os que encontramos, principalmente os importados, são bem pesados. Quero mostrar o início destes assuntos para que você pesquise mais, posteriormente, a fim de se aprofundar. Mesmo assim, o que você absorver aqui já será de grande ajuda em projetos de todos os tamanhos.

2.2 Hipertexto

Como qualquer criança na idade escolar, eu fazia trabalhos escolares o tempo todo. O processo quase sempre era assim: a professora definia o assunto e, ao chegar em casa, eu pegava um bocado de papel almaço (lembra-se?) e ia para casa da minha tia. Lá era onde as enciclopédias ficavam. Se eu precisasse fazer um trabalho escolar, certamente, tais enciclopédias me ajudariam na tarefa.

As enciclopédias eram pesadas. Chatas de serem usadas. Porém, continham tudo ou quase tudo do que eu precisava para completar o meu trabalho.

O que eu gostava nas enciclopédias era a facilidade de encontrar as informações de que eu precisava. Funcionava mais ou menos assim: se eu quisesse saber sobre "automóveis", eu procurava

na primeira enciclopédia. Ao final do texto, sempre havia uma referência para um assunto relacionado, neste caso, por exemplo, sobre "motores", que me levava para uma segunda enciclopédia e assim por diante. Eu sempre conseguia encontrar, rapidamente, o que desejava.

Organizando a informação

A informação – um texto, por exemplo – pode ser organizada de forma linear ou não linear. Imagine um livro. Toda a informação contida no livro está estruturada de forma linear, isso é, há começo, meio e fim. Você não consegue absorver todo o significado de uma história se iniciar sua leitura pela metade do livro. Você precisa, necessariamente, começar lendo o primeiro capítulo até chegar ao último, para absorver e entender o máximo possível de informação sobre a história contada.

As enciclopédias possuem informações organizadas de forma não linear, ou seja, as informações não estão em uma ordem específica, mas sim, de forma relacional e associativa. Exemplo: quando você procura informações sobre "veículos automotores" em uma enciclopédia, ao final do texto, você pode encontrar uma série de referências dos assuntos relacionados, como, por exemplo: "motores de combustão interna", "rodas", "tipos de combustíveis", "mecânica" etc.

Essa maneira não linear de organizar a informação é baseada em como a sua mente funciona. Quando você pensa em um assunto, seu cérebro faz uma série de associações para formar uma ideia, trazer à tona uma memória ou uma lembrança. Por esse motivo, seu cérebro consegue guardar informações que podem ser recuperadas quando pensamos diretamente nelas ou quando pensamos em assuntos relacionados.

Vannevar Bush

Entender sobre como organizar informação é entender como funciona a internet. Estudiosos como Vannevar Bush e Ted Nelson foram os pioneiros em formar ideias sobre como sistematizar grandes volumes de informação, não contendo apenas textos, mas também, imagens, vídeos e sons. Ted Nelson cunhou o termo **Hipermídia**. Ele criou todo o conceito de linkagem de textos, por volta de 1960. Entretanto, muito antes, em 1945, Vannevar Bush descreveu em seu artigo *As We May Think* (http://bit.ly/BushAsWeMayThink) um dispositivo chamado Memex. Este dispositivo armazenaria uma grande quantidade de informações relacionadas e conectadas entre si, possibilitando a recuperação de tais dados de maneira fácil, sem dispender muito trabalho manual.

Se levarmos esse conceito para web, percebemos que apenas uma tag faz esse trabalho de referência e associação de informação: o **link**. Na web, relacionamos informações, sites e qualquer outro objeto usando a tag "A". Quando você linka um texto, uma imagem, um gráfico, você associa esta informação ao destino do link. Você está referenciando assuntos, como na enciclopédia. Nós usamos o link todos os dias e aposto que metade dos desenvolvedores não mensura a importância dessa tag. Sem os links, não há web. Não conseguimos associar nada a coisa alguma. Perde-se a referência.

Sugiro que você pesquise mais sobre a história do **hipertexto**. Você vai entender a amplitude do que significa organizar informação e sua importância para os dias de hoje, tanto para web, quanto para o mundo em si. É um assunto fascinante! Se quiser, comece por aqui: http://bit.ly/History_of_hypertext.

2.3 Marcação

Não adianta linkar e relacionar informações na web se essas informações não detiverem significado.

Você sabe diferenciar um título de um parágrafo porque é um ser humano letrado (assim espero). Você consegue fazer isso porque reconhece as características de cada elemento: o título tem letras grandes e, geralmente, poucas palavras; o parágrafo tem letras pequenas e é formado por muitas palavras. Essas características fazem seu cérebro diferenciar um elemento do outro. Assim é com todos os outros elementos no HTML.

Os meios de acesso como os sistemas de busca, leitores de telas, browsers, scripts ou quaisquer outros que tentem ler seu código HTML não possuem essa inteligência e precisam de ajuda para entender cada um desses elementos. Essa ajuda é o que chamamos de tags. As tags são marcações que formam cada elemento do HTML. Um parágrafo é marcado com a tag `<p>`, um título importante com a tag `<h1>` etc. Cada tag tem sua função e seu significado.

Logo, o HTML serve para **dar significado à informação**, e a isso nomeamos de **semântica**.

Ótimo! Agora que sabemos basicamente o que significa semântica, fica mais fácil entender sobre a estruturação semântica de layouts. As novas tags (nem tão novas assim) do HTML5 foram criadas para que possamos externar a semântica do código, não deixando o significado apenas para seu conteúdo, mas também para a estrutura onde o conteúdo se encontra. Explico melhor: imagine dois títulos na página — um no cabeçalho e outro no rodapé. Você sabe que o título do cabeçalho é mais importante do que o título do rodapé. Mas os sistemas de busca, como o Google, não discernem esse fato. Para ele, são dois títulos e pronto. Um no começo da página e outro no final. Por esse motivo, existem as tags `<header>` e `<footer>`. Dessa forma, em vez de criarmos `<div class="cabecalho">`, podemos criar `<header class="cabecalho">`. Os sistemas de busca ou qualquer outra coisa conseguem saber que a tag `HEADER`, por exemplo, se trata de um cabeçalho. Em consequência disso, é fácil identificar e definir a importância de cada informação de acordo com o local no qual ela foi inserida.

As tags do HTML5 nos deram a possibilidade para acabar com a "sopa de DIVs" que eram os websites e sistemas web. O código ficou organizado visualmente e, definitivamente, mais semântico.

2.4 Microdata

O conceito de **Microdata** não é novo. Se você já ouviu falar sobre Microformats, vai se familiarizar facilmente com microdados.

Para quem ainda desconhece, pense no seguinte: como você diz para algum sistema de busca que um determinado texto no site é uma resenha? Ou, como você disponibiliza as informações sobre seu

evento, como o local e data para serviços de terceiros na internet?

Os Microdados são atributos inseridos nas tags do HTML com informações para complementar seu significado.

A especificação de Microdados do HTML define um mecanismo que permite que meios de acesso como dispositivos, scripts, sistemas, serviços etc. reconheçam dados que possam ser inseridos em documentos HTML com facilidade. Esses dados devem ser legíveis para seres humanos e para máquinas, devendo ser também compatíveis com outros formatos de dados existentes, como `RDF` ou `JSON`.

O Google nos fornece um exemplo muito interessante. Veja a seguir o texto:

```
1 <div>
2   Meu nome é James Bond, mas todos me chamam de Bond.
3   Este é o meu site: <a href="#">example.com</a>
4   Moro em Londres, Inglaterra, e trabalho como detetive secreto.
5 </div>
```

Quando você, ser humano, lê este texto, entende todas as informações. O problema é que as "máquinas" não conseguem interpretar textos (ainda). Por isso, é interessante que indiquemos essas informações para que possamos reutilizá-las em outros projetos. Tais informações complementares são úteis para scripts que criamos todos os dias em nossos sistemas.

Agora, esse mesmo código recheado de Microdados:

```
1 <div itemscope itemtype="http://data-vocabulary.org/Person">
2   Meu nome é <span itemprop="name">James Bond</span>
3   mas todos me chamam de <span itemprop="nickname">Bond</span>.
4   Esta é a minha página inicial:
5   <a href="#" itemprop="url">example.com</a>
6   Moro em Londres, Inglaterra, e trabalho como
7   <span itemprop="title">detetive secreto</span>
8   na <span itemprop="affiliation">ACME Corp</span>.
9 </div>
```

Nesse código, indicamos que o texto se trata de uma pessoa (`itemtype`), cujo nome (`itemprop="name"`) é James Bond, mas o apelido (`itemprop="nickname"`) é Bond. Ele também tem um website (`itemprop="url"`), é detetive secreto (`itemprop="title"`) e trabalha (`itemprop="affiliation"`) na ACME Corp.

A ideia é que a informação seja acessível para qualquer coisa. Qualquer coisa mesmo. Se você consegue ler, uma máquina deve conseguir ler também. Se uma máquina pode ler, ela precisa interpretar esses dados, como seu cérebro faz. Por isso, é importante que uma mesma linguagem seja conversada entre máquina e ser humano, de modo que toda informação pode ser usada para outros

fins.

A ideia de existirem complementos para expandirem o significado de informações em elementos do HTML não é nova. Os *Microformats* já haviam tentado entrar em voga uma vez, sem sucesso. Houve poucos adeptos, e muitos serviços e browsers em potencial simplesmente não adotaram a tecnologia. Agora, com uma nova época no desenvolvimento web, em que diversas necessidades têm surgido, principalmente em uma era na qual dispositivos usam a internet como plataforma, algumas empresas voltaram a se interessar pelo assunto. Um exemplo disso é o trabalho que o Google, o Yahoo, o Bing e a Yandex tem feito com o padrão Schema (http://schema.org), o qual é baseado no padrão de Microdata.

Lembre-se que o objetivo principal é entregar conteúdo para o usuário, não importa como ele esteja acessando essa informação. Se for por meio de um App ou um sistema web, se for por um celular ou por um óculos (não importa!), ele deve conseguir consumir a informação que desejar e, principalmente, poder reutilizá-la.

2.5 W3C e WHATWG ─ Amor e ódio

Se você é novo na área, deve entender que o desenvolvimento para internet passou por várias revoluções. O Tableless nasceu de uma delas. Uma das revoluções mais recentes, e que você deve ter acompanhado de perto, foi o nascimento do HTML5. Na verdade, pouca gente sabe como tudo aconteceu, mas foi bem simples.

O WHATWG (*The Web Hipertext Application Technology Working Group*) é um grupo de desenvolvedores que não se conformavam com a velocidade com que as coisas aconteciam no W3C. Eles são um grupo separado do W3C. Foi fundado por membros da Mozilla, Apple e Opera por volta de 2004. Eles resolveram iniciar a escrita de uma nova recomendação para o HTML, já que o W3C iria abandonar o HTML e focar seus esforços em linguagens baseadas em XML. É por isso que, algum dia no seu passado, você começou a fechar as tags `BR` com uma barra no final (`<br/>`). Sugiro que leia este capítulo do livro *HTML5 For Web Designers* (http://html5forwebdesigners.com/history/index.html#the_schism_whatwg_tf).

Hoje, entendemos que esse seria um caminho bem ruim. Portanto, o WHATWG resolveu iniciar essa documentação de forma independente, de modo que os desenvolvedores e os browsers pudessem avançar o desenvolvimento web com uma tecnologia nova e totalmente focada em inovações. Deu certo. Como os membros tinham influência na comunidade e também dentro dos browsers, o HTML5 logo se tornou popular. Os browsers começaram a adotar as novidades imediatamente e os desenvolvedores começaram a implementá-las em seus projetos.

Contudo, a partir desse momento, iniciou-se uma rixa entre o WHATWG e o W3C. O próprio Tim Berners-Lee, todavia, admitiu em outubro de 2006 (http://dig.csail.mit.edu/breadcrumbs/node/166) que as tentativas deles de dirigir a web do HTML para XML não estavam funcionando muito bem. Foi por esse motivo que o W3C decidiu criar um HTML Working Group, começando do zero e usando as especificações do WHATWG como base para a futura versão do HTML. Isso foi bom e

ruim ao mesmo tempo: o W3C iniciou um grupo que iria trabalhar no HTML 5 (note o espaço antes do 5) e o WHATWG estava trabalhando na versão HTML5 (sem espaços), que seria usada pelo próprio W3C como base!

Um dos problemas foi o seguinte: quando há duas especificações, qual os desenvolvedores devem seguir? Aposto que você, hoje em dia, recorre pouco à documentação do W3C, não é? Normalmente, as informações de novidades no HTML5 vêm de blogs especializados no assunto. Mas de onde esses blogs tiram essas informações? Eu gosto bastante do WHATWG, porque a documentação deles é bastante ativa. Eles escrevem-na baseando-se em pequenos pedaços e não como algo monolítico, que muda de tempos em tempos, depois de muita, muita revisão. Eles são mais ágeis.

Mozilla Chief Technology Officer, Andreas Gal

"Generally we tell developers to look at the WHATWG version, which tends to be developed with better technical accuracy,"

Há um artigo na C|Net muito pertinente que discorre sobre essa picuinha entre os dois grupos. Se você quiser entender mais sobre o assunto, leia: *HTML5 is done, but two groups still wrestle over Web's future* (bit.ly/html5-web-future), escrito pelo Stephen Shankland.

Mas quem decide quais as novidades vindouras? O W3C está trabalhando no HTMl 5.1, que inclui uma série de novidades em Canvas e *drag-and-drop*. Entretanto, o WHATWG também está trabalhando em coisas desse gênero. Há outro problema apontado pelo WHATWG, que alega que o W3C vive copiando suas partes da recomendação e usando em suas documentações oficiais. No artigo que o Stephen escreveu, há um exemplo comentando a especificação sobre os padrões de URL, no qual o W3C possui um rascunho sobre o assunto, mas em alguns momentos, o W3C aponta o rascunho do WHATWG como sendo a última versão.

O W3C tem tentado se mover mais rápido no processo de transformar rascunhos em recomendações. No entanto, isso não é tão fácil assim. Se você se move rápido, você precisa se desapegar de algumas coisas. A ideia do W3C é que tudo seja altamente estável. Não dá para correr e entregar alguma coisa se aquilo, em algum momento, vai travar. Essa é a grande diferença entre o W3C e o WHATWG. Para mim, é bastante compreensível.

Existem três pilares principais: W3C, browsers e devs. E eu sempre presto atenção aos browsers. Se os browsers acham bacana uma determinada especificação, eles começam a adotá-la, de forma que os devs implementam essa novidade em seus projetos. O que sobra para o W3C? Seu papel é bastante importante para regulamentar estes padrões. Não creio que o W3C vá sair de cena ou algo do tipo. Ele é imprescindível para o desenvolvimento da web. Mas precisa seguir a velocidade dos devs, assim como os browsers têm feito. Por isso, é importante a atuação de representantes dos próprios browsers nos grupos de trabalho do W3C. Também que é muito importante que **você** participe das listas de discussão dos grupos de trabalho. Aqui, estão todas as listas de e-mail de que você pode participar do W3C (http://lists.w3.org/).

Vamos ver como isso tudo se desenrola. Participe. Não seja um dev passivo às mudanças.

CSS

Ah, o CSS. Criado em 10 de Outubro de 1994 por Håkon Wium Lie e Bert Bos por conta da necessidade de formatar documentos HTML, formatando decentemente pela primeira vez a informação que era publicada na internet.

A principal diferença do CSS, comparado a outras linguagens parecidas, era a possibilidade da herança de estilos em cascata dos elementos. Isso explica o nome *Cascading Style Sheets*. A ideia é que você não precisa formatar elemento por elemento, mas que cada estilo seja herdado para facilitar e dar liberdade para modificar a formatação da informação. Logo, se você quiser que todos os parágrafos da página sejam da cor azul, basta escrever uma linha e todos os parágrafos ficarão azuis.

3.1 As camadas de desenvolvimento

Vale dar uma explicação rápida de um conceito bastante antigo e que está para mudar, mas que mesmo assim rege a forma com que nos organizamos no código: existem 3 camadas no desenvolvimento web: a informação, formatação e comportamento. A camada de *informação*, normalmente controlada pelo HTML, exibe e dá significado à informação que o usuário consome. É nela que você marca o que é um título, um parágrafo etc., dando significado e estruturando cada pedaço de informação publicada.

A camada de *formatação* é normalmente controlada pelo CSS. É nela que você controla como a informação será formatada, transformando o layout feito pelo designer em código. O CSS é o responsável por dar forma à informação marcada com HTML, para que ela seja bem-vista em qualquer tipo de dispositivo.

Já a camada de *comportamento* era controlada pelo JavaScript. Mas agora o CSS também está tendo alguma responsabilidade nesse terreno. Nesta camada é onde controlamos qual será o comportamento da informação. Quando uma modal se abre ou um slider de imagens funciona, é o JavaScript que está entrando em ação.

Essas são descrições bem básicas de cada camada. Expliquei isso tudo para que você entenda o principal objetivo do CSS, que é controlar a formatação da informação.

Contudo, entenda o seguinte: o CSS nunca vai dar significado a qualquer informação, embora ele possa produzir alguma pseudo-informação na tela usando os *pseudo-elementos :after* e *:before* (marque para estudar depois). O CSS também não é uma linguagem que faz as coisas funcionarem, tipo uma linguagem server-side. Mas isso você já deve saber. O lugar do CSS é bem definido: formatar a informação.

O futuro do CSS ainda é uma incógnita. Durante muito tempo pintamos quadradinhos, literalmente, com CSS. Hoje fazemos elementos 3D e controlamos animações e transições na tela. Nós até substituímos algumas tarefas que antes eram feitas com JavaScript. Mas ninguém sabe ainda como o CSS vai caminhar daqui para a frente. Será que ele vai ganhar uma linguagem secundária para cuidar de animação? Ou será que sua sintaxe precisa ser modificada para contemplar todas as

possibilidades?

São dúvidas com que você não precisa se preocupar agora, mas saiba que daqui alguns anos você não estará digitando mais CSS como fazemos hoje.

3.2 Progressive Enhancement e Fault Tolerance

Não há como falar sobre Progressive Enhancement antes de falar de Fault Tolerance.

Fault Tolerance é como as máquinas tratam um erro quando ele acontece. É a habilidade do sistema de continuar em operação quando uma falha inesperada ocorre. Isso acontece a todo momento com seu cérebro. O sistema não pode parar até que esse erro seja resolvido, logo ele dá um jeito para que esse erro não afete todo o resto. A natureza inteira trabalha dessa forma. Os browsers trabalham dessa forma. É por isso que você consegue testar as coisas maravilhosas do CSS3 e do HTML5 sem se preocupar com browsers antigos.

Por exemplo, quando escrevemos uma propriedade de CSS que o browser não reconhece, ele simplesmente ignora aquela linha e passa para a próxima. Isso acontece o tempo inteiro quando aplicamos novidades do CSS ou do HTML. Lembra-se quando os browsers não reconheciam os novos tipos de campos de formulários do HTML5? O browser simplesmente substituía o campo desconhecido pelo campo comum de texto. Isso é importante porque o que se faz hoje no desenvolvimento de um website, continuará funcionando de alguma forma daqui a 10 anos. Como os browsers têm essa tolerância a falhas, linguagens como HTML e CSS ganham poder para evoluir o tempo inteiro, sem os bloqueios das limitações do passado.

Tudo sobre acessibilidade

Fundamentalmente, Progressive Enhancement é tudo sobre acessibilidade. Na verdade, o termo *acessibilidade* é normalmente usado para indicar que o conteúdo deve ser acessível para pessoas com necessidades especiais. O Progressive Enhancement trata isso mas na ótica de que todo mundo tem necessidades especiais e por isso o acesso deveria ser facilitado para qualquer pessoa em qualquer tipo de contexto. Isso inclui facilmente pessoas que acessam websites via smartphones, por exemplo, cuja tela é pequena e algumas das facilidades que existem no desktops estão ausentes.

Níveis e tolerância

Nós passamos por alguns níveis ao desenvolver algo tendo como método o Progressive Enhacement. Esses níveis têm como objetivo sempre servir primeiro o conteúdo e depois todas as funcionalidades e comportamentos que podem melhorar o consumo deste conteúdo e também de toda a página.

A base para tolerar erros é sempre manter um fallback quando algo ruim acontecer. A primeira camada geralmente é dar um fallback básico, conhecido pela maioria dos dispositivos, para servir um conteúdo em forma de texto. Isso é óbvio porque texto é acessível para praticamente qualquer

coisa. Muitos dos elementos do HTML têm um fallback de texto para casos onde elemento não seja carregado ou não seja reconhecido. Lembra do atributo ALT? Até mesmo nas tags de vídeo e áudio, como a seguir:

```
1 <video src="video.ogg" controls>
2   Texto de fallback.
3 </video>
```

A segunda camada é a semântica do HTML. Cada elemento do HTML tem sua função e principalmente seu significado. Eles acrescentam significados a qualquer informação exibida pelo HTML e muitas vezes estendem o significado que o texto sozinho não conseguiria passar.

O texto é universal

A terceira camada de experiência é a camada visual, onde o CSS e também as imagens, áudios e vídeos são responsáveis. É onde a coisa fica bonita e interativa. Aqui você sente mais a tolerância dos browsers a falhas. Usamos o tempo inteiro propriedades que nos ajudarão a melhorar a implementação de layouts, mas que em browsers antigos podem não ser renderizados. Experimentamos isso a todo momento.

A quarta camada é a camada de interatividade ou comportamento. O JavaScript toma conta dessa parte controlando os elementos do HTML, muitas vezes controlando propriedades do CSS para realizar ações de acordo com as interações do usuário.

A camada final é uma extensão da semântica dos elementos do HTML. Aí é onde inserimos as iniciativas de WAI-ARIA. É onde vamos guiar leitores de telas e outros meios de acesso para elementos e pontos importantes na estrutura em que o layout se baseia, indicando quais regiões e elementos são referência de navegação.

Há outro lado é pensar em *Gracefull Degradation*, que é exatamente o oposto do Progressive Enhancement. O Gracefull Degradation é a forma de fazer seu website degradar harmoniosamente, ou seja, em vez de você pensar em uma base sólida para que todos os browsers antigos consigam renderizar seu site sem problemas, o Gracefull Degradation nivela sempre tudo por cima. A ideia é que você inicie o desenvolvimento fazendo tudo o que há de melhor nas linguagens, mas tentando prever como os elementos ficarão caso um browser não consiga reconhecer, por exemplo, uma propriedade do CSS.

O Gracefull Degradation é uma abordagem um pouco arriscada porque você nunca consegue prever em todos os cenários o que exatamente o browser do usuário vai deixar de renderizar. Por isso, a abordagem do Progressive é algo mais correto, uma vez que você já preparou as possibilidades para caso um problema aconteça.

Na minha opinião, você não precisa se preocupar com isso tudo, basta ter em mente que o cliente precisa ter a melhor experiência que o dispositivo e o software podem dar. Essa é a forma mais segura e mais produtiva de se trabalhar.

3.3 Organizando seus assets ─ Breve explicação

Aprender CSS não é algo complicado. A sintaxe é bastante simples de entender e os resultados são vistos facilmente no browser. Você escreve, atualiza a página e boom, os resultados estão logo ali. Mas quando se trata da organização do código e da estrutura de arquivos, a coisa toda pode complicar dependendo do tamanho do projeto. Embora a sintaxe do CSS seja bastante simples, tudo pode ir por água abaixo se a estrutura dos seus arquivos não forem bem planejados.

Organizando os arquivos

Você tem uma série de maneiras para organizar seu CSS. Mas a maneira mais comum é modularizar os arquivos. Cada arquivo fica responsável por uma parte do site. A decisão que você vai tomar é: dividir os arquivos por áreas do site ou por componentes.

Normalmente, dividimos os arquivos de CSS por áreas quando o projeto é pequeno, por exemplo, em um projeto de website institucional. Você pode criar um arquivo para formatar a sidebar e seu conteúdo. Outro para formatar o header, outro para o footer, outro para a página interna de texto e assim por diante.

Em projetos maiores, como grandes portais de conteúdo ou produtos, é muito melhor separar os arquivos de CSS por módulos e componentes de layout. Você vai ter um CSS que vai formatar todos os botões do site, outro para tipografia, outro CSS vai formatar listas, o menu, as listas de notícias da sidebar etc. Em sites maiores, você vai criar mais páginas combinando estes elementos.

Claro, há exceções. Mas geralmente essas duas formas de organizar seu CSS cobrem todas as suas necessidades. A estrutura de pastas é de gosto pessoal. EU (portanto, é o jeito como eu gosto de trabalhar) separo uma pasta na raiz do projeto chamada `assets`. Dentro dela, terei outras duas com o nome de `JavaScripts` e `stylesheets`. Você pode chamar essas pastas de `js` e `css` se achar melhor. Tem gente que não gosta do nome da pasta no plural… De novo, gosto pessoal.

A pasta de imagens pode ou não ficar dentro dessa pasta `assets`. Eu costumo colocá-las dentro.

Uma boa ideia é fuçar em projetos alheios pelo GitHub para encontrar uma estrutura de que você goste mais ou que você entenda que seja a mais comum. Mas o importante é manter seu código organizado.

3.4 Organizando o código

Se você trabalha com uma equipe, o código escrito precisa parecer que foi produzido por uma pessoa. Essa é a regra de ouro em projetos com equipes pequenas ou grandes. Você precisa transitar pelo projeto todo e sentir que aquilo foi escrito por você. Para tanto, cada um da equipe precisa

escrever o código do projeto seguindo uma série de padrões de escrita, para que no final, o resultado seja algo homogêneo.

O @mdo (twitter.com/mdo), um dos criadores do Twitter Bootstrap, criou um guia de regras de codificação bastante interessante, e eu concordo com a maioria das regras. Seria bastante interessante que você lesse depois na íntegra — http://diegoeis.github.io/code-guide/ — e tentasse apresentar essas regras para seus colegas de trabalho. A produtividade vai ser notada logo nas primeiras semanas.

Aqui vão algumas das dicas:

- Atributos HTML devem usar esta ordem em particular para facilitar a leitura do código: class, id, data-*, for, type ou href, title, alt, aria-*, role.
- Sempre que possível, evite elementos pais supérfluos quando escrever HTML. Muitas vezes é necessário uma interação ou refatoração, mas isso produz pouco HTML.
- Assegure-se rapidamente de que a renderização do seu conteúdo declare o encoding de caracteres explicitamente: meta charset="UTF-8".
- Use soft-tabs com dois espaços.
- Quando agrupar seletores, deixe seletores individuais em uma linha.
- Inclua um espaço antes de abrir um colchete.
- Coloque o fechamento do colchete em uma nova linha.
- Inclua um espaço depois dos dois pontos `:` em cada declaração de propriedade.
- Cada declaração deve estar em sua própria linha para encontrarmos melhor os erros.
- Feche todas as declarações com ponto-vírgula `;`.
- Valores separados por vírgula devem incluir um espaço logo depois da vírgula (ex. `box-shadow`).
- Não inclua espaços depois das vírgulas, como em cores `rgb()`, `rgba()`, `hsl()`, `hsla()` ou `rect()`. Isto ajuda a diferenciar múltiplos valores (vírgulas, sem espaço) de cores de propriedades múltiplas (vírgulas, com espaço). Também não coloque valores iniciados com zero (ex. `.5` em vez de `0.5`).
- Deixe todos os valores hexadecimais em caixa baixa: `#fff`. Letras com caixa baixa são mais fáceis de serem lidas e entendidas quando queremos scanear o documento.
- Use cores em hexadecimal abreviadas quando puder, por exemplo: `#fff` em vez de `#ffffff`.
- Coloque aspas nos valores em seletores com atributos: `input[type="text"]`.
- Evite especificar unidades quando os valores forem 0: `margin: 0;` em vez de `margin: 0px;`.

3.5 CSSOM

O CSSOM é um assunto pouco falado no mercado. Não que não seja importante, porque é, mas ele não afeta diretamente sua produtividade. É um daqueles assuntos teóricos que diferencia um bom dev de um ótimo dev front-end. Por isso, eis minha preocupação em fazer você conhecer esse assunto.

O HTML tem a DOM, e o CSS tem a CSSOM, acrônimo de *CSS Object Model*. Quando o browser está carregando o DOM do HTML, que é a árvore de objetos do HTML, ele encontra um chamado para um código CSS e começa a montar outra árvore, baseada nos seletores do CSS. Essa árvore faz uma referência entre os seletores, que vai representar os elementos no HTML e suas propriedades visuais que serão modificadas pelo CSS.

O Google tem uma imagem muito bacana que nos ajuda a explicar:
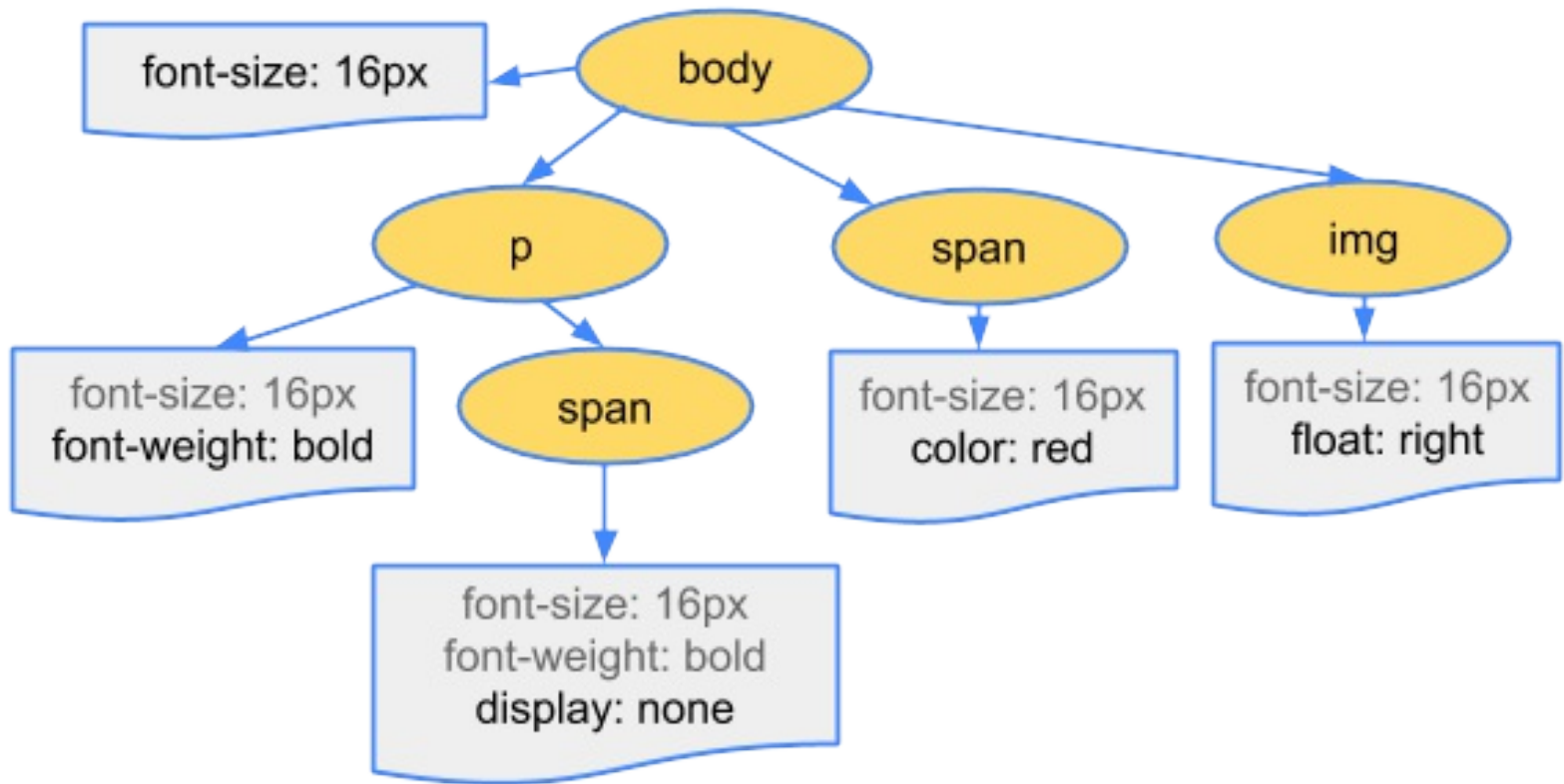
Fig. 3.1: Representação da árvore do CSSOM.

O Google tem um artigo bem legal que fala sobre o CSSOM: http://bit.ly/1tENlgY

O browser começa a aplicar os estilos pelo seletor mais geral e depois vai substituindo essa herança de acordo com os seletores mais específicos.

```
1 body {
2  color: red;
3 }
4
5 article p {
6  color: blue;
7 }
```

Se você define que a cor do texto do body é red, o browser define que todo o texto que ele encontrar dentro do body terá a cor vermelha. Mas lá na frente você diz que o parágrafo de dentro da tag article é azul, ele vai substituí-lo por este estilo. É assim que o browser executa seus estilos, por isso a

forma de árvore do CSSOM.

Simples, mas bastante interessante, não?

3.6 Um pouco de CSS 3D

O CSS 3D é sem dúvida uma das features do CSS mais aguardadas por todas as crianças do Brasil. Fala a verdade! Fazer efeitos com sombra, gradientes, transparências etc. um dia já foi algo bacana na vida do desenvolvimento. Mas fazer com que os elementos se comportem em um espaço 3D é sensacional. O CSS 3D traz para a web efeitos visuais nunca antes visto.

Mas não se anime muito. Eu sei que você está ansioso para sair por aí colocando efeitos 3D de CSS em tudo quanto é aplicação. Mas calma... Entenda que o CSS foi feito para estilizar documentos. Você o utiliza para melhorar a experiência dos usuários nos diversos dispositivos e não para enfeitar seu website como se fosse uma penteadeira. Lembra-se dos websites cheios de gifs animados? Pois é, cuidado para não cair no mesmo erro. Você estará utilizando o CSS 3D da maneira certa se seus efeitos passarem despercebidos pelo usuário ao utilizar seu sistema. Encher seu sistema com efeitos a cada clique ou a cada ação pode fazer com que o usuário perca tempo e a paciência.

Tudo é uma questão de perspectiva

Para falar de 3D, precisamos falar sobre perspectiva. Para manipular uma área 3D o elemento precisará de perspectiva. Você pode aplicar a perspectiva ao elemento de duas formas: utilizando diretamente a propriedade `perspective` ou adicionando um valor `perspective()` na propriedade `transform`.

```
1 div {
2   perspective: 600;
3 }
```

ou

```
1 div {
2   transform: perspective(600);
3 }
```

Prefixos de browsers

Os exemplos não usam os famosos prefixos de browser para não complicar o exemplo. Por isso, para fazer seus testes, use os prefixos `-webkit-`, `-moz-` e `-ms-` antes das propriedades.

Estes dois formatos são os gatilhos que ativam a área 3D onde o elemento irá trabalhar.

O valor da perspectiva determina a intensidade do efeito. Imagine como se fosse a distância de onde vemos o objeto. Quanto maior o valor, mais perto o elemento estará, logo, menos intenso será o visual 3D. Logo, se colocarmos um valor de 2000, o objeto não terá tantas mudanças visuais e o efeito 3D será suave. Se colocarmos 100, o efeito 3D será muito visível, como se fosse um inseto olhando um objeto gigante.

É interessante que você também leia um pouco sobre o ponto de fuga (http://bit.ly/1wG2R2P). O ponto de fuga por padrão está posicionado no centro. Você pode modificar essa posição com a propriedade `perspective-origin`, que é muito parecida com a propriedade `transform-origin`, que define onde a ação de transformação do objeto acontecerá, nesse caso quando falamos de ações 2D. A propriedade `perspective-origin` afeta os eixos X e Y do elemento filho.
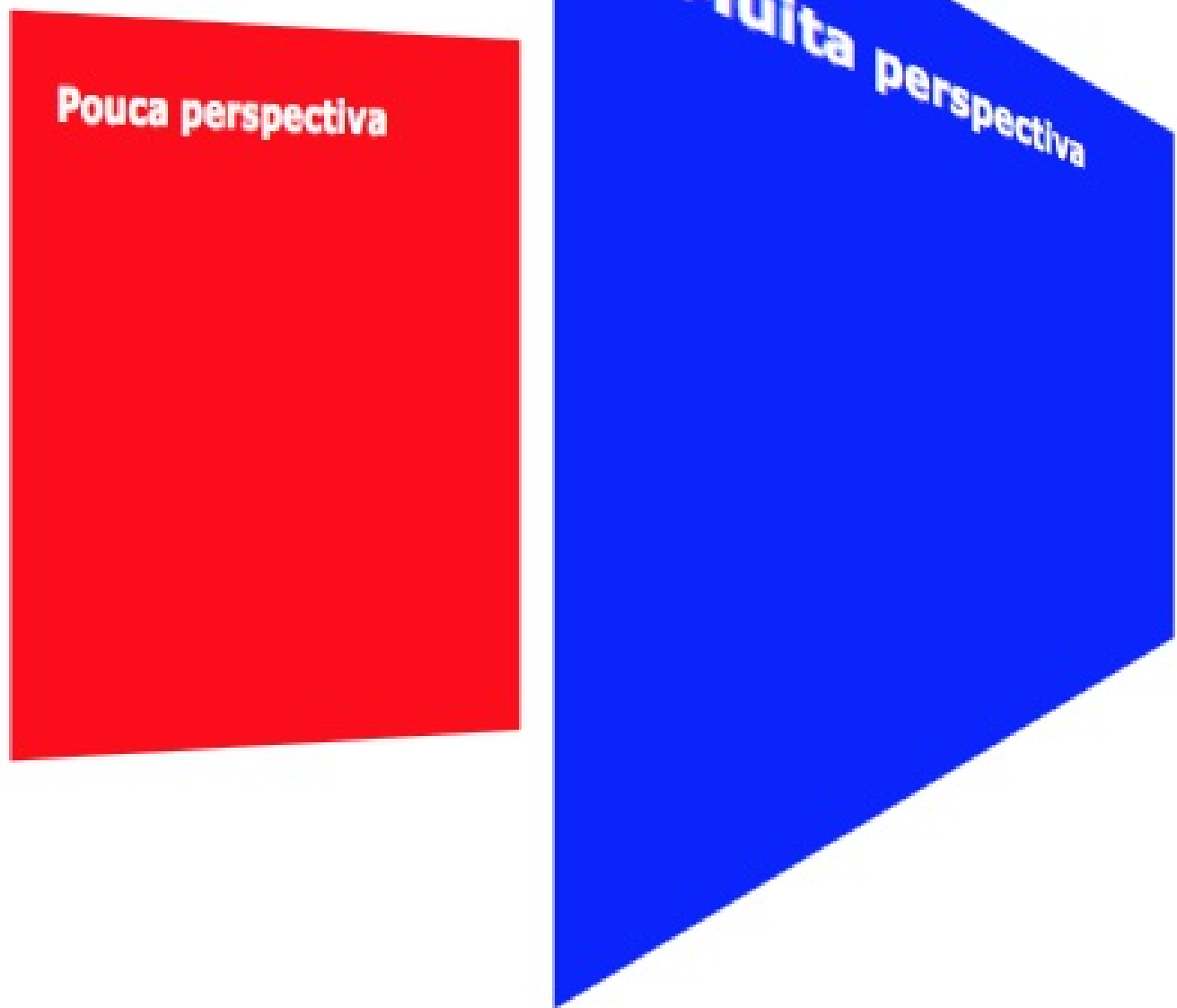
Fig. 3.2: Exemplos de perspectiva usando CSS3D.

CSS 3D Transforms

Você deve estar acostumado a trabalhar com os eixos X e Y no CSS padrão. No CSS 3D podemos manipular também o eixo Z, que representa a profundidade.

Veja na imagem logo a seguir, estou utilizando os valores `rotateY`, `rotateX` e `translateZ`. Perceba que no `translateZ` eu utilizei valores negativos e positivos. Quando utilizo o valor negativo, o objeto fica "mais longe" e, se coloco valores positivos, o objeto fica "mais perto".
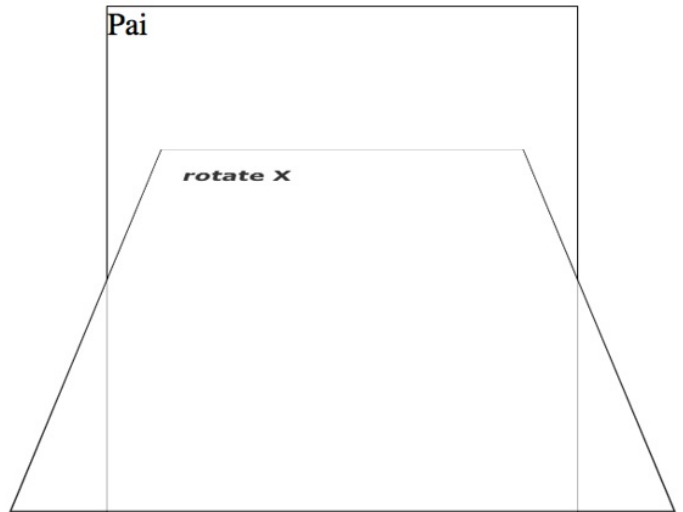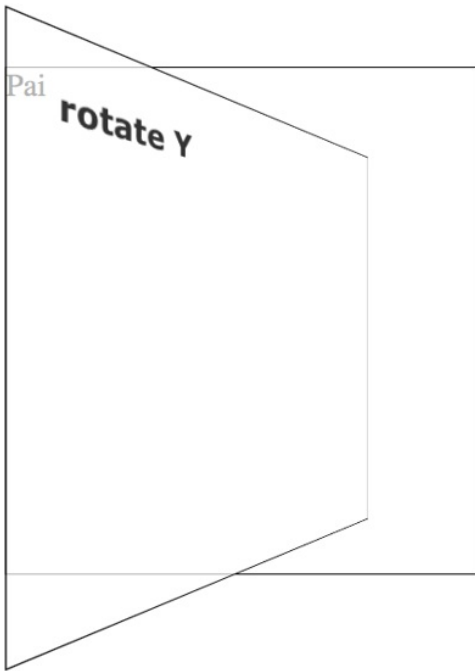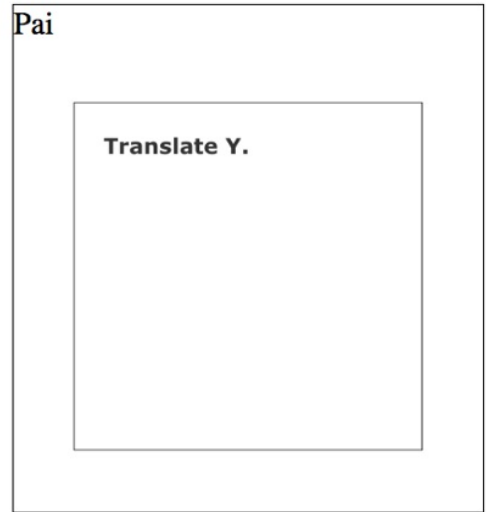
Fig. 3.3: Diferenças do CSS transform com os valores de rotate e translate.

```
1 .pai {
2   border:1px solid #000;
3   padding:0;
4   float:left;
5   margin:50px 100px;
6 }
7
8 .pai div {
9   width:200px;
```

```
10   height:200px;
11   color:#333;
12   font:bold 15px verdana;
13   padding:20px;
14   background:rgba(255,255,255,0.7);
15   border:1px solid #000;
16 
17   perspective: 500;
18 }
19 
20 .pai div.translateY {
21   transform: translateZ(-100px);
22 }
23 
24 .pai div.translateZ {
25   transform: translateZ(100px);
26 }
27 
28 .pai div.rotateY {
29   transform: rotateY(45deg);
30 }
31 
32 .pai div.rotateX {
33   transform: rotateX(45deg);
34 }
```

Nós podemos utilizar também alguns atalhos para esses valores onde podemos definir as três dimensões de uma vez:

- translate3d(x,y,z)
- scale3d(x,y,z)
- rotate3d(x,y,z,angle)

Muito importante: ao utilizar as propriedades resumidas, os browsers ativam automaticamente a aceleração por hardware no Safari para que as animações tenham uma melhor performance.

3.7 Sobre Frameworks CSS

Se você trabalha em um grande projeto onde há elementos diferentes e vários comportamentos, talvez seja uma ótima ideia criar um framework para esse projeto. Normalmente, eu crio pequenos frameworks para projetos específicos, geralmente produtos. Dificilmente crio frameworks quando estou fazendo websites, mas alguns princípios ainda se mantêm e isso é muito bom porque, embora

não haja um framework por trás, há um alto nível de organização de código, que o torna legível para manutenções posteriores.

Por que criar um framework?

Existem diversos motivos para você criar um framework para seu projeto. Alguns deles são:

Prototipação

Para provar uma teoria, é importante que possamos criar protótipos de tela ou de um fluxo inteiro do projeto. É importante que os protótipos tenham a interface muito fiel à interface final. É diferente quando você pede para o usuário testar determinada tela se ela ainda está com cara de wireframe. Nesse caso, o framework ajuda bastante. Tendo todos os módulos e os comportamentos prontos, você e qualquer um da equipe conseguem um fluxo de telas em algumas horas, mesmo não sendo um desenvolvedor front-end.

Usando o framework para fazer esse protótipo, você tem a garantia de que o visual será exatamente o que irá para a produção e o usuário não sentirá aquela estranheza de interface inacabada. Não há a necessidade nesse ponto de ter qualquer back-end feito. Você está na fase de prototipação e por isso tudo com o que o usuário interagir será fake, resumindo: serão apenas páginas estáticas, com links simples que levam para outras telas.

Padronização

Um framework não cria apenas padronização de interface, mas padronização de código e comportamentos. É muito importante que o usuário utilize de forma uniforme o produto inteiro. O usuário se acostuma com os elementos, seus funcionamentos e comportamentos. Todos esses pontos juntos formam um padrão de uso importante para que o usuário não se perca durante o fluxo.

Produtividade e manutenção

Se o código de todos os módulos já existe, a construção de uma nova tela é muito rápida. Não é necessária a ajuda de designers o tempo inteiro já que há um padrão de interface. A manutenção de código é mínima já que você consegue controlar todos os módulos de forma individual.

Se os back-ends precisam fazer pequenas modificações na tela, com uma documentação bem feita, eles conseguem se virar sem a ajuda de front-end ou do time de UX. Isso é importante para que exista o mínimo de burocracia na rotina da equipe.

Por que não criar um framework?

Embora haja bons motivos para criarmos nosso framework, há também cenários ruins que você precisa relevar:

Layouts diferentes

Não aconselho a criar um framework para um website institucional, um hotsite ou um site pequeno. Websites têm telas muito diferentes umas das outras, não há maneira de criar um framework que se adeque a todos os designs. Por isso, se as telas do projeto são totalmente diferentes, sem repetição de elementos, sem padrão visual, a criação de um framework pode trazer mais problemas do que soluções.

Código que não para de crescer

Para grandes projetos, você precisa ter um planejamento para prevenir a quantidade de código que seu framework gera. Se seu framework é tão grande que começa a prejudicar a performance do sistema, algo está errado. Seu legado não deve estar sendo bem gerenciado e você pode estar mantendo código morto no projeto.

Tempo inicial

Para construir um framework do zero, você precisará ter o tempo da sua equipe e talvez de outras equipes para decidir a ideia inicial do framework, como: quais problemas ele irá resolver, quais tecnologias serão usadas, padrão de código, padrão visual vindo do time de UX, como os back-ends usarão o framework, como será a documentação entre outros pontos importantes. Isso além do tempo gasto inicialmente para codificar a base do framework e todo o desenvolvimento posterior.

Já existem outros frameworks no mercado

Tente estudar as opções que já existem. Eu sei que muito provavelmente nenhum framework disponível irá se adequar ao design do seu projeto, mas tente considerar que talvez você não precise do framework inteiro; às vezes utilizar apenas o GRID de um determinado framework seja o suficiente para o seu projeto.

3.8 Sobre Style Guides

Embora vários Style Guides sejam apresentados como frameworks, um Style Guide tem uma ideia um pouco diferente. Um Style Guide é apenas um guia para ser seguido por diversas equipes. Eles não têm necessariamente elementos e módulos prontos, mas têm uma documentação bem acabada e definida de diversos pontos como tipografia, grid, cores, dimensões e medidas. Mas, como eu disse, não há um módulo pronto de botões, há as especificações de como um botão será.

Geralmente, os Style Guides são feitos pelos designers ou pelo time de UX. São eles que detêm os PSDs, os Wireframes e outras informações sobre a interface. Este é um ponto importante a se considerar. Existem equipes de UX que não fazem essa documentação e em muitos casos esse é o problema de interfaces mal acabadas, com tipografia tosca e outros problemas.

Veja por exemplo o *brand toolkit* da Mozilla (bit.ly/mozilla-styleguide), que mostra como equipes e parceiros devem tratar o brand e o estilo visual dos elementos de websites ligados à empresa.

Estilos de design e padrões de código são uma maneira importante de se certificar de que a interface será consistente. Muitas vezes, do Style Guide, surgem frameworks bem acabados e completos.

Quer um exemplo? Veja o Style Guide feito pelo pessoal da BBC: http://www.bbc.co.uk/gel

Exemplos de frameworks usados em empresas

Uma série de grandes empresas tem seus próprios frameworks. São com eles que a consistência de layout e interface conseguem perdurar entre equipes e diversos projetos internos. Confira alguns frameworks:

- *Locaweb Style* é um framework feito pela equipe que eu coordenei na Locaweb. Ele contempla todos os projetos internos da Locaweb e os produtos usados por todos os clientes. Como é um framework open source, também pode ser usado por qualquer um fora da Locaweb — http://locaweb.github.io/locawebstyle/
- *Yelp!*: http://www.yelp.com/styleguide
- *The Guardian*: https://github.com/guardian/frontend
- *Style Guide e Design Standards* do setor de transporte público de Londres: bit.ly/london-transporte-design-standards
- *West Virginia University*: http://brand.wvu.edu/
- *Style Guide e Design Patterns* da IBM: http://www.ibm.com/design/language/index.shtml
- *Apple Human Interface Guidelines*: bit.ly/apple-human-interface

Segue um site com exemplos, artigos e livros sobre Style Guides e frameworks: http://styleguides.io/

3.9 Leitura recomendada

Continue seus estudos com os links a seguir:

- Introdução ao CSS 3D — Exemplo de Card Flip — http://tableless.com.br/introducao-ao-css-3d-flip-card/
- CSS modular: construindo front-ends flexíveis e escaláveis — http://www.infoq.com/br/presentations/css-modular
- Modulando o CSS — http://tableless.com.br/modulando-o-css/
- Princípios para escrever CSS de forma consistente — http://tableless.com.br/principios-para-escrever-css-de-forma-consistente/
- 6 estratégias para melhorar a organização do seu CSS — http://tableless.com.br/6-estrategias-para-melhorar-a-organizacao-do-seu-css-2/

Um pitaco sobre editores

Editores são a principal ferramenta dos desenvolvedores. É no seu editor que você vai passar boa parte do seu dia. Mesmo assim, muitos desenvolvedores não gostam de perder uma ou duas horas aprendendo sobre sua principal ferramenta de trabalho. É aí que muitos perdem a oportunidade de se tornarem produtivos e ágeis nas tarefas diárias.

Entender como seu editor funciona é essencial. Saber quais suas limitações, até onde você pode estender suas funcionalidades, conhecer quais plugins facilitam o seu processo de trabalho etc. Por isso, quando escolher seu editor do coração, passe um tempo com ele, aprendendo seus meandros e conhecendo seus truques.

Não importa qual editor você vai usar. Eu vou expor minha opinião sobre os editores WYSIWYG na próxima seção, mas essa é minha opinião. Se você quiser usar um editor assim, não há problema nenhum, contanto que você tenha controle sobre seu código final. A regra básica é: quem manda no código é você e não seu editor.

4.1 Esqueça os editores WYSIWYG

Quero deixar claro que não sou contra Dreamweaver. Eu até usava quando estava iniciando na área. O problema do Dreamweaver é que ele não ajuda no aprendizado. Ele vira uma muleta com o passar do tempo e você vira um desenvolvedor mal acostumado. Isso faz você parar no tempo.

O Dreamweaver de certa forma ajudou bastante o desenvolvimento web. Quando as tabelas se tornaram populares, o modo WYSIWYG do Dreamweaver era imprescindível. Como ninguém fazia websites decentes, escrever tabelas na unha era um parto. O Dreamweaver, por sua vez, facilitava todo o processo em tempos de tabelas aninhadas, com código inútil e websites com obesidade mórbida.

Durante muito tempo, o Dreamweaver formou gerações e gerações de desenvolvedores e empresas. Era realmente o editor que mataria o Front-Page... Que Deus o tenha. O Dreamweaver conseguiu ser um dos editores mais populares na sua época. Ele era realmente completo.

Com a profissionalização da área, mais e mais desenvolvedores começaram a escrever seus códigos manualmente e o Dreamweaver começou a perder mercado. Qualquer um que ficasse um pouco mais esperto percebia que, apesar das facilidades, com o tempo o código automático se tornava um pé no saco e a manutenção, impossível. Até então, todo o processo de atualização de código (leia-se FTP) ajudava o Dreamweaver a ficar na liderança. Eu ouvia diversos desenvolvedores dizendo que o Dreamweaver era matador simplesmente por causa da facilidade de acessar o FTP e mudar o código dos arquivos direto no servidor. Até hoje lembro do medo que isso me causava.

Editores mais simples e mais poderosos que o Dreamweaver surgiram. Homesite e EditPlus eram os meus prediletos. A escolha do editor de texto é totalmente pessoal e por isso eu não vou ficar chateado se você não gostar da minha indicação aqui, que é o Sublime Text. Mas é minha obrigação dizer que o Sublime Text é usado pela maioria dos devs atualmente até o ano de 2015.

Sendo mais específico: o Dreamweaver é um símbolo. Qualquer editor ou ferramenta que vire uma muleta para o dev é algo ruim. Você precisa digitar e entender de cabo a rabo o significado do código escrito. Se você tem uma ferramenta que faz tudo sozinha, você não é um desenvolvedor, você é apenas um operador de software.

Procure sempre produzir seu código manualmente. Se você está começando e tem algum editor como o Sublime Text, que fica autocompletando tudo o que você escreve, desabilite essa função, pelo menos por enquanto. Isso vai obrigá-lo a pesquisar, a procurar a resposta, vai forçá-lo a lembrar daquele negócio de que você esqueceu. Isso vai lhe fazer pensar.

Não, isso não quer dizer que você precisa aprender VIM para poder se transformar em um dev cool. Longe disso. O VIM é muito legal, mas sua curva de aprendizado é bastante longa. Você até vai se tornar bastante produtivo com ele, mas acho que se você está iniciando nessa área, não comece pelo VIM.

4.2 Sublime Text

O Sublime Text é hoje o editor mais popular entre qualquer um que digite código. Não porque ele é simples ao máximo, mas por causa da facilidade de personalização de funções e interface.

Há uma quantidade enorme de plugins para Sublime que ajudam a melhorar a produtividade do seu desenvolvimento, seguem alguns deles:

- *Package Control* é um gerenciador de plugins para Sublime. Você consegue procurar e instalar plugins diretamente pelo seu editor, sem ter que baixar arquivos externos nem nada. Aconselho instalá-lo primeiro antes de instalar os outros plugins. Vai facilitar sua vida.
- *GitStatus* monitora mudanças de arquivos e status do arquivo no repositório do projeto.
- *Emmet* dá suporte ao famoso e velho ex-Zen Coding.
- *EditorConfig* faz o Sublime reconhecer o `.editorconfig` com as configurações de formatação dos arquivos.
- *SideBarEnhancements* coloca uma série de novas opções no menu contextual da sidebar facilitando o gerenciamento de arquivos direto do editor.
- *Sass* dá suporte a highlight de sintaxe do Sass para Sublime.
- *AngularJS* traz snippets para quem gosta de AngularJS.

Existe uma série de outros plugins no Package Control que podem ser instalados agora no seu Sublime, além de várias outras funcionalidades que o transformam em um editor bastante poderoso e produtivo.

Suas preferências

Uma das primeiras coisas que você precisa saber é que o Sublime guarda suas preferências em um arquivo em formato JSON, por isso é bem fácil de entender. Os dois arquivos que você precisa

saber: um para seus atalhos de teclado, que fica em `Sublime Text 2 > Preferences > Key Bindings – User` e outro para as definições gerais, em `Sublime Text 2 > Preferences > Settings – User` ou `CMD + ,`.

Nesses dois arquivos você muda o que quiser. Se você tiver que mudar qualquer coisa, lembre-se de mudar sempre no arquivo de `USER`, que é o seu arquivo. Assim você mantém as definições default intactas.

Uma sacada é colocá-los dentro da sua pasta de Dropbox, no GIT ou em qualquer outro lugar, de forma que você consiga recuperá-los quando formatar a máquina ou quando quiser configurar o Sublime em outro computador. Eu coloquei os meus dentro de uma pasta no Dropbox e fiz um link simbólico para o Sublime.

Se você quiser descobrir o nome de um determinado atalho referente a um comando de teclado, basta abrir o console do Sublime com o atalho ``CMD + ` `` e a cada comando de teclado que você executar, você verá o nome do comando referente a este atalho.

Lista de comandos

Sublime tem muitos comandos escondidos que podem não estar listados nos menus. Você pode acioná-los pelo controle de acesso `CMD + SHIFT + P`. Lindo, não?

GoTo Anything

Quer encontrar funções, classes ou IDs dentro dos seus arquivos? Use o atalho `CMD + P`. Esse comando abre o search geral do Sublime. Nesse search você pode procurar arquivos, como no exemplo anterior, ou se você iniciar a busca com um `@`, vai possibilitar a procura de funções, classes, IDs etc. no arquivo aberto.

Se você quiser ir para uma linha específica, você pode começar a busca com `:` (dois pontos) e colocar o número da linha, ou ainda use o atalho `CTRL + G` e número da linha.

Não vou passar todas as dicas sobre Sublime Text aqui, isso merece um livro ou post gigante. Mas você pode encontrar muita coisa útil nesses links:

- http://simplesideias.com.br/configurando-o-sublime-text-2
- http://tableless.com.br/7-plugins-sublime-text-que-voce-deveria-conhecer/
- http://tableless.com.br/dicas-truques-sublime-text/
- https://github.com/fnando/sublime-text-screencasts

4.3 Outros editores

Sim, eu indico outros editores além do Sublime Text, são eles: Brackets e Atom.

Brackets by Adobe

O Brackets é um editor criado pela a Adobe, para fazer parte do seu novo pacote de ferramentas para desenvolvimento chamada Adobe Edge. Só que a Adobe, ao contrário da sua antiga cultura, disponibilizou o código-fonte sob licença MIT. Desde então o código aberto é encontrado via GitHub e é mantido pela comunidade. O Brackets é escrito em HTML, CSS e JavaScript, o que é bastante interessante.

Brackets é meu segundo editor predileto. Embora eu o ache um pouco mais lento que o Sublime Text, as features que ele vem adotando, principalmente em parceria com a Adobe, são maravilhosas e totalmente focadas em produtividade.

Uma das features que tem o dedo da Adobe é o Extract, um serviço do Creative Cloud e que no Brackets é algo nativo. A ideia é simplesmente assim: você sobe um PSD para o Extract, ele abre o PSD em uma interface direto do seu browser. Você clica nos elementos de texto, cor etc. e o serviço lhe passa o código CSS. O código é bastante enxuto. Sou bastante contra coisas que fazem códigos front-end de forma automática, mas este está indo no caminho certo. Mesmo assim você pode usar esse serviço sem utilizar o Brackets.

Leia mais sobre o Brackets:

- Site oficial — http://brackets.io
- Extract for Brackets — http://www.adobe.com/br/creativecloud/extract.html
- GitHub do projeto — https://github.com/adobe/brackets

Atom by GitHub

O Atom foi feito pelo pessoal do GitHub e o seu principal apelo é a integração entre o editor e o serviço GitHub.

É quase impossível não comparar o Atom com o Sublime. O Sublime cravou um padrão com seus shortcuts, sua interface e sua coleção de comandos que é muito difícil ignorar. O Atom tem a cara do Sublime, diferente do Brackets, que é bem diferente tanto na interface quanto nos shortcuts. Faço uso do Brackets para algumas coisas e estou gostando bastante. Mas a curva de aprendizado, quando acostumado com o Sublime, é muito grande. Com o Atom você já vai conhecer meia dúzia de shortcuts, facilitando muito a migração.

O Atom seguiu o mesmo caminho. Ele pegou tudo que era bom do Sublime e ainda teve ideias originais e geniais para facilitar ainda mais a nossa vida. É como se o Atom fosse uma nova versão do Sublime. É difícil até de pensar o que o Sublime mudaria para melhorar e até concorrer com o Atom.

Essa similaridade entre os editores não é ruim. Mas o que vai importar mesmo são as features extras

que cada um pode trazer no futuro. Descrevi algumas features legais que me chamaram mais a atenção e que se diferenciam totalmente do Sublime.

No Atom, a personalização da interface é muito mais fácil, já que é feita via CSS. Sim, CSS. Seguindo o menu `Atom > Open Your Stylesheet`, o Atom abre um arquivo CSS onde você customiza o que quiser na tela. Muda cor de texto, font, background, margin, padding, Tudo. Aí vem a pergunta: mas como vou saber os elementos que devo formatar? Simples, abra um Inspector. Siga o menu `View > Developer > Toggle Developer Tools`. O Inspector que se abre é o padrão do Webkit.

Uma coisa de que sinto falta no Sublime é uma parte de configurações onde eu possa mudar o que eu quiser no Editor. O Atom tem uma área dessa, muito bem acabada e organizada. No Sublime você muda essas preferências direto nos arquivos de configuração do editor. Isso é genial na primeira vez, mas depois você sente alguma falta de apertar botões, saber que tudo está funcionando, ter certeza de que não alterou nada que vá destruir seus shortcuts, tema e outras configurações. Já o Atom tem essa tela de configurações, como qualquer outro programa. Nele, você gerencia os packages (plugins), temas e shortcuts.

O interessante no Atom é que tudo é tratado como módulos. O Atom chama seus plugins de Packages. Estes packages fazem parte das configurações do editor ou são apenas plugins que estendem as funcionalidades já existentes. Todos esses packages são módulos independentes e estão no Github, prontos para serem forkados, compartilhados etc.

Seguem dois links que explicam essas e outras coisas mais no Atom.

- Atom, o novo editor do GitHub — http://tableless.com.br/atom-o-novo-editor-github/
- Primeiras impressões do Zeno com o Atom — http://www.youtube.com/watch?v=5BwwUnJVIIs

Cuidando do seu código

5.1 Esqueça o FTP

Simplesmente esqueça. Ninguém mais usa FTP para atualizar seu código em produção.

O protocolo de FTP é algo muito simples e durante muito tempo cumpriu bem seu trabalho. Mas a partir do momento em que você precisa gerenciar bem o legado de código do seu sistema, ele passa a não ser uma boa alternativa por vários motivos.

Entenda que o seu fluxo de deploy deve ser seguro. Algumas empresas burocratizam o processo, outras tentam simplificar o máximo possível, mas todas devem se ater a segurança de fazer rollback ou atualizar o código em produção sem perigos e, se possível, de modo automatizado.

Como você faz o deploy de um site inteiro com FTP? Não importa o tamanho do site, você nunca sabe exatamente qual arquivo foi modificado. Na dúvida, você acaba copiando todos os arquivos, para não ter erro. Isso tudo manualmente. Alguns clientes de FTP tentam fazer uma sincronização comparando data de modificação e tamanho de arquivo. Mas geralmente essa comparação demora muito e não é confiável.

Há também a necessidade de, por algum motivo, você precisar voltar ao cenário anterior caso algo dê errado na transferência dos arquivos. Se você não fez um backup antes de copiar os arquivos, você vai ter um problemão.

Usar um controle de versão é algo indispensável nos dias de hoje. É por isso que eu digo pra você: pare de usar FTP e aprenda GIT ou qualquer outro sistema de controle de versão como SVN ou Mercurial. Eu e grande parte dos devs preferimos GIT. Hoje ele é padrão de mercado e muito provavelmente você vai esbarrar com ele nas empresas em que você trabalhar.

5.2 Sobre controles de versão

Você não precisa trabalhar em uma equipe grande para saber que há um problema para controlar o código do seu projeto. Para resolver isso existem sistemas que controlam seu código, monitorando cada mudança e juntando todas as modificações de forma que não ocorram conflitos quando duas pessoas editam a mesma linha de um mesmo arquivo.

Durante muito tempo, o controle de versão mais conhecido era o SVN (Apache Subversion). Isso mudou depois que o GIT, criado por Linus Torvalds, chegou. Existem outros controles de versão como o Bazaar ou o Mercurial, mas o que você precisa saber mesmo, até o dia em que estou escrevendo este texto, é o GIT.

O básico sobre qualquer controle de versão

Este é o básico que você precisa saber para entender qualquer controle de versão. A partir daqui, você estará seguro para saber pelo menos o funcionamento básico de qualquer controle de versão até então.

Branchs e Trunks

A última revisão, a mais atual, normalmente é chamada de *HEAD*. Existem momentos onde teremos vários *HEADs* pois o projeto tem vários *BRANCHES*. Um branch é uma cópia do projeto. Cada branch pode ser editado e evoluído independentemente.
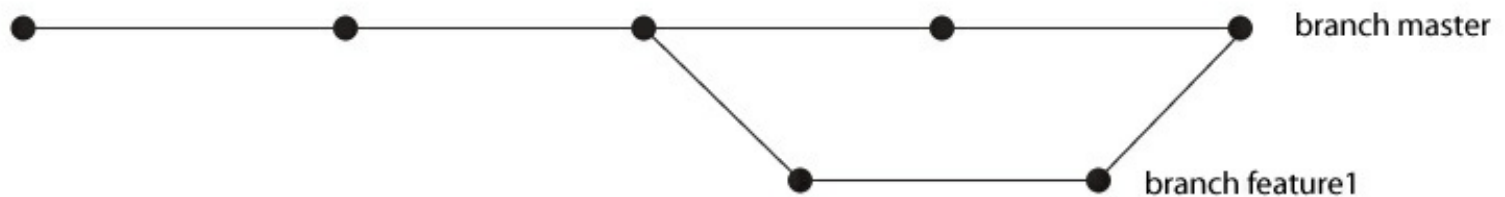
Fig. 5.1: Branchs são como galhos de uma árvore.

Imagine que exista a versão de produção, que é aquela que está no ar. É a versão que o usuário está utilizando. Não é a versão mais atual, pois a mais atual é a que está na sua máquina, mas a versão que está em produção é (ou deveria ser) a versão mais estável. Essa versão está numa branch, normalmente, chamada `MASTER`. Nem sempre se chama Master, depende da equipe. Há pessoas que chamam de Prod, por exemplo.

Versionamento

Versionamento é uma das características mais básicas do controle de código. Quando comecei a desenvolver, normalmente eu trabalhava guardando pastas com nomes do tipo *nomeDoProjeto-v1*, *nomeDoProjeto-v2* etc. Estas pastas guardavam apenas as versões com grandes modificações de projeto, algo como mudança geral de layout, uma feature importante ou algo do tipo. O problema são as pequenas edições. Quando juntamos as pequenas modificações, temos uma grande modificação. É muito provável que, se você perder essa versão, você não se lembre de todas as pequenas modificações feitas e isso é muito inconveniente. Eu não tenho saudades dessa época.

Em um sistema de controle de versão, toda vez que o desenvolvedor termina uma determinada tarefa, ele gera um *commit*. O commit é onde o desenvolvedor atrela as modificações do código com uma explicação exata do que ele fez naquela tarefa. Cada commit gera uma versão do seu código. A graça é que se você tem versões do código, praticamente um undo gigante do seu projeto. Se depois de uma determinada modificação, de qualquer um da equipe, seu projeto começou a apresentar problemas, basta voltar as versões até encontrar exatamente em qual dos commits os problemas iniciaram. Você

tem um log completo dos passos do projeto.

Eu confesso que no início eu achava muito, mas muito chato ter que escrever commits com explicações sobre as mudanças que fiz naquela tarefa. Eu achava que estava perdendo tempo, que tudo era muito burocrático e chato. Mas depois que percebi o valor nesse processo, comecei a escrever commits mais elaborados. O que nos leva ao próximo tópico: Log.

Log

A cada commit feito, o desenvolvedor adiciona uma mensagem explicando o que ele fez naquela tarefa. É importante que essa mensagem seja clara e rica em detalhes, mas não um texto gigante, apenas o suficiente para que você mesmo ou outras pessoas envolvidas entendam o que aquela submissão significa.

Isso pode servir como documento depois, facilitando que novos integrantes da equipe entendam o histórico do projeto e para que a equipe inteira esteja ciente das modificações feitas pelos colegas.

Existem equipes que levam tão a sério as descrições dos commits, que o log do projeto se torna uma verdadeira documentação do projeto. Você consegue entender quais foram os passos de cada desenvolvedor, quais os problemas resolvidos e quais soluções aplicadas.

Diffs

Os Diffs são comparações das modificações feitas no código, mostrando todas as alterações, deleções, inserções etc. É como a gente compara duas versões de um mesmo código. Veja um exemplo direto do GitHub:

```
2  source/assets/javascripts/locastyle/_general.js                                View
@@ -42,7 +42,7 @@ locastyle.general = (function() {
42    function _autoTrigger(){                                    42    function _autoTrigger(){
43      var hash = window.location.hash.replace("!/#", "");       43      var hash = window.location.hash.replace("!/#", "");
44      if(hash !== ''){                                          44      if(hash !== ''){
45  -     $('[data-target=' + hash + '], a[href=' + hash +        45  +     $('[data-target="' + hash + '"], a[href="' + hash +
    ']').trigger('click');                                            '"]').trigger('click');
46      }                                                         46      }
47    }                                                           47    }
48                                                                48
```

Fig. 5.2: Visualizando um Git Diff no app do GitHub.

Há também a possibilidade de fazer um Diff via console. Eu, particularmente, não gosto. A interface é bastante ruim e confusa. Se você utilizar serviços como GitHub ou BitBucket, prefira as versões de Diff deles. Veja a seguir como se parece o Diff feito pelo GIT direto pelo terminal:

```
diegoeis:locastyle (wip) $ git diff fab5f14c1c93d5de6b 0ac9eff087a9a
diff --git a/source/assets/javascripts/locastyle/_general.js b/source/assets/javascripts/locastyle/_general.js
index c91f719..b7521dd 100644
--- a/source/assets/javascripts/locastyle/_general.js
+++ b/source/assets/javascripts/locastyle/_general.js
@@ -42,7 +42,7 @@ locastyle.general = (function() {
   function _autoTrigger(){
     var hash = window.location.hash.replace("!/#", "");
     if(hash !== ''){
-      $('[data-target="' + hash + '"], a[href="' + hash + '"]').trigger('click');
+      $('[data-target=' + hash + '], a[href=' + hash + ']').trigger('click');
     }
   }
```

Fig. 5.3: Git Diff direto pelo terminal. Coisa feia, né?

Merge

Lembro-me de equipes inteiras perdendo arquivos e partes de código pois dois membros abriam o mesmo arquivo para editar. Obviamente nunca, mas nunca dava certo. Isso fazia com que grandes equipes criassem mecanismos da idade das pedras para avisar o resto do pessoal que determinado arquivo estava sendo usado. O trabalho remoto nunca era possível. Com o controle de versão isso muda. Qualquer um pode editar qualquer arquivo a qualquer hora.

O sistema de controle de versão compara as modificações feitas no arquivo pelos desenvolvedores e junta os códigos automaticamente. Muito bonito, hein?

E quando os desenvolvedores modificam a mesma linha de código? Aí o controle de versão gera um conflito, que deve ser resolvido pelo desenvolvedor que enviou as modificações por último. Funciona assim:

1. Dois desenvolvedores abrem o mesmo arquivo para editar;
2. Quando eles terminam, cada um cria um commit. Se os desenvolvedores fizeram edições em partes diferentes do arquivo, por exemplo, um no começo do arquivo e o outro no final, o sistema junta os códigos fazendo um merge e criando um arquivo atualizado com as duas revisões. Indolor. Caso contrário:
3. O sistema percebe que os desenvolvedores fizeram as modificações na mesma linha, o sistema gera um conflito. Isso é necessário porque, obviamente, o sistema não vai saber qual dos dois códigos é o correto;
4. O conflito é resolvido pelo desenvolvedor que fizer o commit do código por último. Eu commitei minha versão de código e subi essa atualização para o servidor. Quando você commitar seu código, você é obrigado a baixar a versão mais atual antes de enviar a sua. Nesse ponto o controle de versão avisará que há um conflito em uma parte do seu código. O conflito é resolvido mostrando algo mais ou menos assim:

```
1 p {
```

```
2 <<<<<<< HEAD:style.css
3    color: red;
4 =======
5    color: blue;
6 >>>>>77971db4580b80ae27e8d65caf5208086:estilo.css
7 }
```

A primeira linha é o seu código. A segunda linha mostra a modificação do outro dev. Aquele hash maluco no final é a identificação do commit do outro dev. Assim fica fácil identificar o exato commit no servidor.

Bom, nesses casos você precisa ver qual código é o correto, e algumas vezes isso significa perguntar para seu colega que diachos é aquela coisa que ele fez.

Git não é GitHub

Eu sei que você já deve saber disso, mas já vi tantos iniciantes na área confundirem, que eu decidi colocar um pequeno *disclaimer* aqui: Git não é GitHub!

O GitHub é um serviço que ajuda os desenvolvedores a gerenciar seu código usando o Git como ferramenta. Geralmente o Git é utilizado principalmente via terminal. Mas ter uma interface é interessante para gerenciar o acesso dos desenvolvedores aos projetos, verificar Diffs, consultar histórico dos arquivos etc. Existe uma série de outros serviços como o GitHub, um bastante popular é o BitBucket.

Por isso, não confunda jamais: Git é o sistema de gerenciamento. GitHub, BitBucket e outros, são apenas serviços online que facilitam o gerenciamento de repositórios Git.

Conclusão e referências

Eu fiquei muito tentado a ensinar GIT aqui, mas decidi não fazê-lo. Este não é o intuito deste livro, lembra? Mas o assunto é fascinante e com certeza, usar qualquer controle de versão, seja GIT, SVN ou qualquer outro, só vai trazer benefícios para sua equipe ou para seu projeto pessoal. Cada controle de versão tem suas particularidades, mas as instruções aqui expostas servirão para todos os controles de versão.

Seguem alguns links de referência para começar seus estudos:

- Micro Tutorial de Git (http://www.akitaonrails.com/2008/4/3/micro-tutorial-de-git)
- Screencast — Começando com GIT (http://www.akitaonrails.com/2010/08/17/screencast-comecando-com-git#.U55NHRY6GfE)
- Iniciando com GIT — Parte 1 (http://tableless.com.br/iniciando-no-git-parte-1/)
- Iniciando com GIT — Parte 2 (http://tableless.com.br/iniciando-no-git-parte-2/)
- Contribuindo para projetos Open Source (http://tableless.com.br/contribuindo-em-projetos-open-source-com-o-github/)

- Livro: Controlando versões com Git e GitHub (http://www.casadocodigo.com.br/products/livro-git-github)

JavaScript

Você pode dividir as linguagens front-end em 3 (até agora): HTML, CSS e JavaScript. Durante muito tempo, o JavaScript esteve longe dos holofotes. Muitos programadores não se aprofundavam o suficiente na linguagem, mas os que faziam se apaixonavam. Como o HTML e o CSS, o JavaScript passou por momentos ruins na guerra dos browsers. Mas sempre foi uma linguagem pioneira.

Como várias linguagens, o JavaScript tem várias soluções para um único problema. Ser uma linguagem flexível tem suas vantagens e desvantagens, por isso é importante que você leia muito, muito sobre patterns e os truques que a linguagem pode guardar.

Eu não quero mostrar para você neste capítulo quais APIs, funções e técnicas você deve aprender. Mas abordei dois assuntos importantes: jQuery e Patterns. Muitos novatos nem pensam em estudar Design Patterns do JavaScript quando estão iniciando.

O JavaScript ganhou tanto foco nos últimos anos que surgiram diversas iniciativas. Algumas delas trouxeram novas ideias e melhorias para a linguagem em si, outras foram apenas fogo de palha. O importante é que a linguagem evoluiu muito. A versão ES6 está prestes a ser lançada oficialmente no momento que este livro é escrito, com importantes novidades.

Na minha opinião, o que fez com que pessoas olhassem o front-end de forma diferente foi o JavaScript. De uma hora para outra, uma série de iniciativas surgiram e de repente todo mundo concordou que o JavaScript era uma linguagem maravilhosa. Principalmente os devs back-end, que achavam que a nossa área era só escrever HTML e pintar quadradinhos com CSS.

O Node.js abriu os olhos de muita gente que não dava nada para esse mercado, trazendo grandes oportunidades. Você não precisa se aprofundar nesse assunto agora, mas é bom saber que alguma hora você vai usar algo baseado em Node.js, seja um gerador de site estático (que abordamos aqui neste livro também) ou um framework como Backbone. O Node.js é um interpretador de JavaScript que permite rodar código JavaScript no servidor. Isso é bom e ruim. É muito polêmico ainda escrever JS no servidor. Empresas, principalmente brasileiras, ainda estão perdidas quanto a isso.

Meu conselho aqui é: não queira aprender nada se você não vai usar agora. Claro, não estou dizendo que não precisa estudar, pelo contrário. Você vai estudar coisas mais imediatas, que vão melhorar sua produtividade. Se você não vai usar Backbone agora, só aprenda se *quiser (ou precisar)*.

Além disso, um dev front-end não é só aquele que escreve JavaScript. Envolve muito mais. Na verdade esse é outro problema. Para mim não existe front-end, existe designer e programador. Mas esta é outra história.

Se você for novato, não fique afobado. Você não precisa aprender Backbone ou qualquer outra coisa para se tornar um desenvolvedor front-end. Essa área cresceu tanto que há espaço para todo mundo. É claro que você não vai escapar de escrever JavaScript, você não precisa saber tudo, mas precisa entender o básico.

6.1 Aprenda pelo menos o básico de JavaScript

É difícil entender qual é este básico que você precisa aprender. Tudo depende do seu objetivo. Se você tem uma tendência mais para o design, aprender o básico talvez signifique aprender o suficiente para manipular eventos e comportamentos do DOM. Se você está mais para ser um programador, o seu básico significa aprender muito sobre orientação a objetos, JS no servidor, patterns etc.

Tudo vai depender também de onde você trabalha. Se você trabalha em uma agência web, suas necessidades de JavaScript são diferentes de uma empresa que faz produto para web. Talvez na agência seja interessante se focar em comportamentos e animação, já que você vai fazer peças de layouts interativos. Em empresas de produto, você vai precisar entender mais sobre arquitetura, talvez um pouco sobre a linguagem server-side utilizada, como manipular bem arquivos `json`, fazer `ajax` de forma decente. São diferentes necessidades.

Por isso, não queira aprender tudo que aparece na sua frente. Esteja bem atento ao contexto da empresa em que você trabalha e principalmente no projeto atual. Mesmo assim, existem alguns pontos que são comuns em qualquer lugar que use JavaScript. Tentei abordar aqui um pouco sobre Escopo de variáveis e bem pouco (mesmo) sobre Patterns do JavaScript, mostrando apenas o Module Pattern, que é bem fácil de entender.

6.2 Escopo de variáveis

Variáveis é uma das primeiras coisas que alguém aprende quando está começando a programar. Elas estão em todas as linguagens de programação e muitas vezes, embora sejam fáceis de entender, causam confusão para quem está iniciando, principalmente quando se trata de escopo.

Vou tentar ser o menos técnico possível aqui, mas não dá para mostrar como funciona escopo de variáveis sem ter que mostrar um pouco de código.

Imagine as variáveis como caixinhas. Cada caixinha dessa tem um nome que você deu e vai guardar um determinado tipo de dado. Como em uma caixa normal, você pode substituir seu conteúdo na hora em que quiser. O problema é que você não pode usar uma variável se ela estiver fora do contexto, ou como podemos falar, fora do seu escopo. Existem dois tipos de escopo: global ou local.

Quando uma variável é criada fora das funções da sua aplicação, ela é chamada de *variável global*. A variável global sempre estará disponível para ser usada por toda a aplicação e por qualquer função criada.

Quando você tem uma função, que faz uma tarefa específica, e dentro ela há uma variável, essa variável tem o que chamamos de *escopo local*, e poderá ser acessada apenas por essa função.

Segue um código bem sem vergonha para mostrar como funciona essa coisa toda. Primeiro, veja o exemplo de variável global:

```
1 var varGlobal = 'Variável Global’;
2
3 console.log(varGlobal);
4
5 // log: “Variável Global”
```

Agora, em um escopo de função:

```
1 var varGlobal = 'Variável Global’;
2 console.log(varGlobal);
3
4 function umaFuncao() {
5   var outraVar = 'Uma outra variável’;
6   console.log(outraVar);
7 }
8 umaFuncao();
9
10 // log: “Variável Global”
11 // log: “Uma outra variável”
```

Ótimo, até aqui sem problemas. Se eu tentar chamar um `console.log(outraVar)` fora da função `umaFuncao()`, veja o que acontece:

```
1 var varGlobal = 'Variável Global';
2 console.log(varGlobal);
3
4 function umaFuncao() {
5   var outraVar = 'Uma outra variável';
6   console.log(outraVar);
7 }
8 umaFuncao();
9
10 console.log(outraVar);
11
12 // log: “Variável Global”
13 // log: “Uma outra variável”
14 // log: outraVar is not defined
```

O browser não consegue encontrar o `outraVar` porque ela é uma variável local, seu escopo está apenas dentro do contexto da função.

Mas há um porém: perceba que sempre antes de nomear as variáveis eu coloquei a palavra mágica `var`. Quando definimos isso em uma variável dentro de uma função, estamos explicitando que aquela variável é local. Se eu retirar o `var` de dentro da função, seu escopo passa a ser global. No código a

seguir eu simplesmente retirei o `var` da frente da variável:

```
1 var varGlobal = 'Variável Global';
2 console.log(varGlobal);
3
4 function umaFuncao() {
5   outraVar = 'Uma outra variável';
6   console.log(outraVar);
7 }
8 umaFuncao();
9
10 console.log(outraVar);
11
12 // log: "Variável Global"
13 // log: "Uma outra variável"
14 // log: "Uma outra variável"
```

Isso é bastante perigoso porque, em um código muito grande, onde toda a equipe está trabalhando junto, o valor da variável pode ser modificada por qualquer função do projeto. Eu posso fazer uma variável com o mesmo nome da sua e isso pode causar um problema gigante. Por isso a regra é: sempre trate suas variáveis em escopo local.

Esse é o caminho inicial e básico para estudarmos formas para trabalharmos com encapsulamento no JavaScript. O Jean Carlo Emer, um grande dev brasileiro, falou disso em um dos artigos que escreveu no Tableless (http://tableless.com.br/modularizacao-em-JavaScript/). Acho que é um bom estudo para você se aventurar posteriormente.

Para ler mais:

- http://tableless.com.br/modularizacao-em-JavaScript/
- http://tableless.com.br/elevacao-ou-JavaScript-hoisting/
- http://mzl.la/14hAWep
- http://JavaScriptbrasil.com/2013/10/11/escopo-de-variavel-e-hoisting-no-JavaScript-explicado/

6.3 Module Pattern JavaScript

Design Patterns não é um tópico exclusivo do JavaScript. É um assunto que envolve qualquer linguagem de programação. Muitos que iniciam seus estudos no JavaScript sofrem porque não entendem como a organização da estrutura de código influencia em seus projetos. É uma questão importante que exige um nível bem baixo de conhecimento sobre escopo e orientação a objetos.

Utilizar um Design Pattern no JavaScript define uma estrutura padrão, organizada, por onde o código

do projeto vai crescer. Existe uma série estilos de patterns. Não é só um. Tenha em mente que, acima de tudo, além de um padrão de funcionamento, o Pattern determina um padrão de escrita de código. Ele estabelece como toda uma equipe vai desenvolver seu código, com restrições, mantendo uma mesma identidade.

No livro chamado *Learning JavaScript Design Patterns* (que você pode ler grátis aqui: http://addyosmani.com/resources/essentialjsdesignpatterns/book/), Addy Osmani (addyosmani.com) diz que existem 3 benefícios ao seguir um Design Pattern, em qualquer linguagem, sendo eles:

- Patterns provam soluções.
- Patterns podem ser facilmente reutilizados.
- Patterns podem ser expressivos.

Quando olhamos um pattern, você encontra uma estrutura e um vocabulário que ajudam a expressar uma solução de forma elegante. Não é só escrever, fazer funcionar e pronto. É escrever de forma que não apenas você, mas qualquer um, anos depois, entenda como aquele determinado problema foi solucionado e por que foi solucionado daquela forma.

Mesmo assim, ter um pattern definido no seu projeto não quer dizer que ele é uma solução exata. Você poderia ter resolvido esse problema com outro pattern e outra estrutura.

De cara, isso parece ser complicado. Mas não é. Aprender um pattern JavaScript pode lhe ensinar mais sobre a linguagem do que muito cursinho por aí.

Module Pattern

Eu costumo usar em meus projetos um padrão chamado Module Pattern. Ele é bastante simples de entender e é muito flexível para projetos de todos os tamanhos. Ele é um pattern bastante usado em projetos como jQuery, Dojo e outros.

O Module Pattern é o básico para iniciar seu entendimento sobre encapsulamento e modularização em JavaScript. O interessante desse pattern é que ele reduz o escopo global de variáveis. Por isso será muito difícil ter problemas recorrentes de escopo de variáveis globais em projetos grandes ou com muitos integrantes na equipe. Essa característica permite criar contextos privados, onde as variáveis e funções são acessadas apenas de dentro e não de fora da função principal. Sua utilização é bastante simples. Não vou entrar nos mínimos detalhes pois alguns conceitos (closures, funções anônimas etc.) podem ser complicados para quem está começando, mas há links no final do texto que você pode seguir para entender mais sobre seu funcionamento.

Perceba o código a seguir:

```
1 var contador = (function() {
2   var placar = 0;
3
4   function mudaPlacar(valor) {
```

```
5      placar += valor;
6    }
7    return {
8      aumenta: function() {
9        mudaPlacar(1);
10     },
11     diminui: function() {
12       mudaPlacar(-1);
13     },
14     valor: function() {
15       return placar;
16     }
17   }
18 })();
```

Tudo sempre em inglês

Eu usei os nomes das funções em português nestes e em outros exemplos para que você tenha facilidade de entender a explicação. Mas a melhor prática é codificar tudo em inglês, ou seja, é importante que nomes de variáveis, funções e até comentários sejam escritos em inglês.

O `contador` é uma variável local que armazena um função anônima, que se será autoexecutada assim que o documento terminar de carregar. A função `mudaPlacar` e a variável `placar` são privadas. Apenas as funções que estão dentro do `return` são públicas e podem ser acessadas por outras por meio da função `contador`, por exemplo: `contador.aumenta()`

Primeiro, crie um arquivo `.js` e coloque o código anterior. Depois crie um arquivo `HTML` e coloque o link desse arquivo `JS`. Feito isso, abra esse arquivo `HTML` no seu navegador. Agora vamos fazer um teste rápido. Digite no console `contador.`. Veja que ele vai mostrar todas as funções permitidas para execução.

```
> contador.aumenta
```
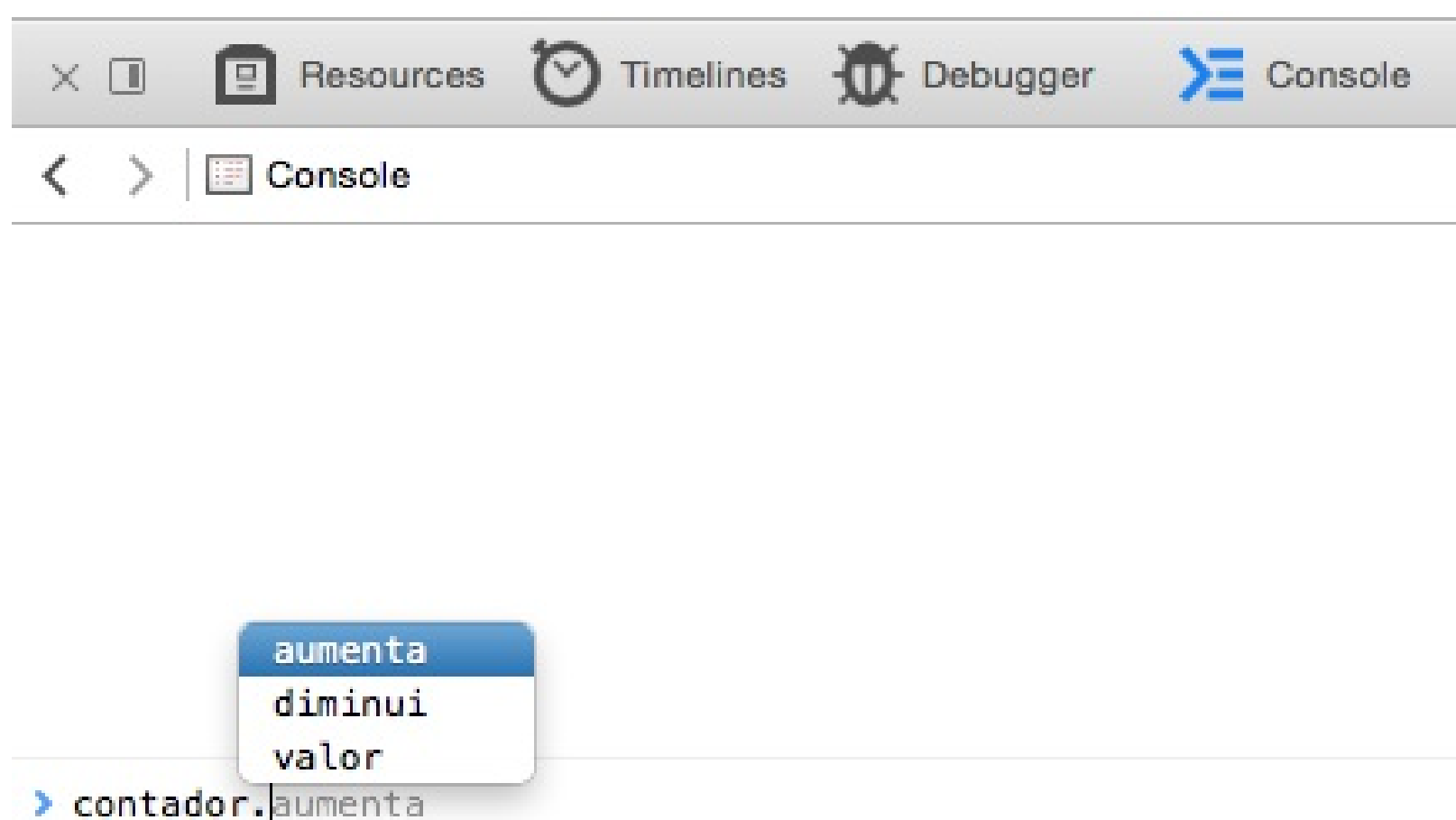
Fig. 6.1: *Acessando as funções permitidas pelo escopo.*

```
> contador.valor()
< 0
> contador.aumenta()
< undefined
> contador.valor()
< 1
>
```

Fig. 6.2: *As funções são executadas normalmente.*

Perceba que a função `mudaPlacar()` não pode ser acessada ou a variável `placar` não pode ser alterada a não ser por intermédio das funções `aumenta`, `diminui` e `valor`.

Se mudarmos um pouco a forma como escrevemos, entramos em um outro tipo de pattern chamado *Revealing Module Pattern*. Nesse caso, todas as funções e valores do módulo são acessíveis no escopo local. Fica assim:

```
 1 var contador = (function() {
 2   var placar = 0;
 3 
 4   function mudaPlacar(valor) {
 5     placar += valor;
 6   }
 7 
 8   function aumenta() {
 9     mudaPlacar(1);
10   }
11 
12   function diminui() {
13     mudaPlacar(-1);
14   }
15 
16   function valor() {
17     return placar;
18   }
19 
20   return {
21     aumenta: aumenta,
22     diminui: diminui,
23     valor: valor
24   }
25 })();
```

Esse pattern é uma boa maneira de trabalharmos com módulos no JavaScript. Modularizar funções e suas dependências é uma forma de reduzir uma série de problemas complexos que temos no momento do desenvolvimento. Nós separamos melhor as funções e seus interesses. A manutenção também fica bastante simples e concentrada.

Eu sei que o Module Pattern pode introduzir de uma vez vários contextos que você ainda não conhece. Mas relaxe. Não precisa entender tudo agora. Vá com calma. A ideia aqui é mostrar que existe e que pode ser usado por você agora em projetos de todos os tamanhos.

Para ler mais:

- Artigo do Jean Carlo Emer: Modularização em JavaScript — http://tableless.com.br/modularizacao-em-JavaScript/
- Artigo bem simples do Nando Vieira: Module Design Pattern — http://simplesideias.com.br/design-patterns-no-JavaScript-module
- http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html
- http://bit.ly/1EV1YHa
- https://carldanley.com/js-module-pattern/
- http://briancray.com/posts/JavaScript-module-pattern
- http://addyosmani.com/resources/essentialjsdesignpatterns/book/#modulepatternJavaScript

6.4 Aprenda jQuery

jQuery é e sempre será um marco na linguagem JavaScript. Mas entenda: jQuery **não** é JavaScript, mas sim uma biblioteca. Ele abstrai a linguagem. É muito fácil encontrar desenvolvedores que confundem jQuery com JavaScript. Esse é o primeiro passo: jQuery não é JavaScript. Nunca diga que sabe JavaScript se você só aprendeu jQuery. Ah, e nunca aprenda jQuery antes de aprender um pouco de JavaScript.

A jQuery ganhou seguidores porque solucionava o problema de compatibilidade entre os browsers. Querendo ou não, o JavaScript é uma linguagem que sofre bastante com a compatibilidade entre browsers. A jQuery conseguiu quebrar essa barreira, fazendo com que tarefas comuns fossem entendidas por todos os browsers e ainda facilitando a sintaxe de escrita.

Animações, manipulação do DOM, incluir e remover classes e muitas outras pequenas tarefas corriqueiras nunca foram tão fáceis de fazer. Mas essa facilidade tem seu preço. Muita gente pulou o aprendizado básico do JavaScript indo direto para a jQuery. Você precisa entender quando está usando a linguagem original ou a biblioteca. É importante que você saiba o limite onde acaba o JavaScript e começa a jQuery.

Eu escrevi um artigo um pouco polêmico no Tableless um tempo atrás com o título *Considere não usar jQuery* (http://tableless.com.br/considere-nao-usar-jquery/), que fala sobre você considerar o uso de JavaScript puro em funções simples ou em websites onde a carga de JavaScript não será muito grande e que muito provavelmente não haverá problemas de compatibilidade entre browsers.

Eu não aconselho a não usar jQuery em grandes sites. Nesse caso, a produtividade que você ganha é gigante e o apoio do time é de 100%. Por isso, saiba dosar. Para projetos pessoais, se você preferir se aventurar, vá fundo, mas na vida real, eu aconselho o uso de jQuery sem pestanejar.

E por que jQuery e não outra biblioteca como o MooTools? Simples: comunidade e popularidade. Praticamente todos os desenvolvedores conhecem jQuery. Ele é muito popular e é usado por mais da metade dos websites da internet (http://blog.jquery.com/2014/01/13/the-state-of-jquery-2014/).

Para ler mais:

- http://youmightnotneedjquery.com/
- http://tableless.com.br/considere-nao-usar-jquery/
- http://simplesideias.com.br/redescobrindo-o-JavaScript-com-jquery
- http://api.jquery.com/

Web Mobile

Os problemas que temos hoje ao desenvolver front-end não mudaram muito da época em que eu comecei, há mais de uma década atrás (ok, me senti um mala e um velho agora). Mas as variáveis com que eu me preocupava para construir um projeto eram resumidamente:

1. Resolução da tela: basicamente se a resolução era 640x480, 800x600 ou 1024x768.
2. Compatibilidade de Browser: basicamente com a família do IE5, 5.5 e 6. As vezes o IE para Mac. Mozilla, Firefox (que foi chamado de Phoenix no começo) e Opera não eram uma preocupação.

E só.

Eu praticamente ignorava o pessoal de 640x480, fazendo websites com largura mínima de 768, para caber com folga nas resoluções acima de 800x600. Além disso eu me lascava para fazer o site funcionar nessas versões do IE. Basicamente essa era a rotina.

Atualmente você precisa se preocupar com esses pontos, mas tudo ficou bem mais complexo. Parte dos browsers se atualiza quase que semanalmente. Até mesmo o Internet Explorer, que era um exemplo da preguiça, está fazendo lançamentos mais regulares de suas versões e cada vez mais compatíveis com os padrões. As resoluções são mais numerosas do que antigamente e parece que isso só tende a aumentar, já que as empresas nunca vão parar de fabricar novos aparelhos.

Quando as primeiras preocupações com os browsers mobile surgiram, os celulares tinham telas tão pequenas e conexões tão ruins que era impraticável fazer qualquer coisa que realmente funcionasse com uma experiência agradável. Passou-se muito tempo até que os mobiles se transformassem em uma prioridade. Mesmo assim, o Opera foi o destaque daquela época dando uma grande ajuda para o mundo inteiro ao lançar o Opera Mini, que foi o primeiro browser mobile decente. Rodava lindamente no meu Nokia 6111, que tinha uma resolução de 128x160. O Opera tentava dar um jeito na bagunça toda, para melhorar a forma como navegávamos na internet em aparelhos com telas no-touch e, principalmente, em telas menores que o buraco de fechadura.
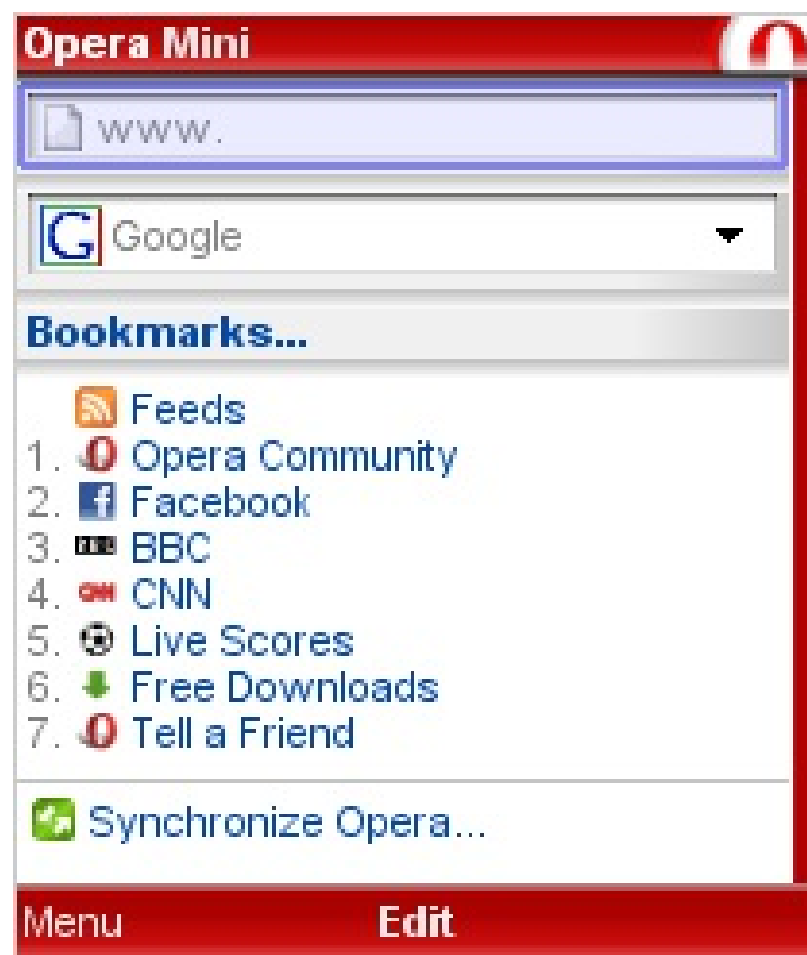
Fig. 7.6: Que saudades desse tempo!

Claro, tudo mudou quando o primeiro iPhone chegou. Seguindo a mesma linha do Opera, o Safari levou a navegação em mobiles para outro nível. Mas se você achou maravilhosa a ideia da Apple, ao mostrar o site em miniatura no Safari, saiba que isso foi ideia original do Opera. Na verdade, os browsers incorporaram várias ideias do Opera, como abas, por exemplo.

Depois que o iPhone se tornou um marco, a corrida do ouro começou e, claro, as preocupações só aumentaram, já que não havia nenhum padrão decente para desenvolvermos para mobile. O W3C fazia uma série de recomendações, mas a única coisa que podíamos fazer até então era formatar o layout para celulares usando a `media type` para *handheld*.

A base do desenvolvimento web mobile

Este capítulo vai mostrar uma série de assuntos que são a base para construir websites que serão bem-vistos em qualquer tamanho de tela. São a base do *Responsive Web Design*, que é a ponta do iceberg de uma coleção de metodologias chamada *Adaptive Web Design*. Entenda que o Responsive Web Design (*RWD*) não é suficiente para fazermos websites prontos para diversos tamanhos de telas. Existem uma série de preocupações com que você vai precisar se importar com o passar do tempo, conforme seu projeto vai crescendo ou seus clientes vão se diversificando. Eu fiz uma apresentação (http://www.slideshare.net/diegoeis/o-bla-bla-bla-do-responsive) sobre isso que pode lhe ser bastante esclarecedora.

7.1 O que é o Adaptive Web Design

Os princípios do desenvolvimento mobile têm se aglutinado em um termo chamado *Adaptive Web Design*. Alguns desenvolvedores confundem este termo com *Responsive Web Design*, mas não se engane: o Responsive é uma das soluções que compõem o termo Adaptive Web Design.

Criar um website adaptativo tem se tornado cada vez mais complexo. São várias metodologias e técnicas que muitas vezes fazem com que essa tarefa de produzir sites mobile se pareça mais complicada do que ela é. De acordo com o *Brad Frost* (http://bradfrostweb.com/), você pode resumir o Adaptive Web Design em 5 pontos:

1. Ubiquidade
2. Flexibilidade
3. Performance
4. Enhancement (de aprimoramento, melhoramento)
5. Future-Friendly

Ele tem um artigo muito, muito importante (http://bradfrostweb.com/blog/post/the-principles-of-adaptive-design/) que explica exatamente cada um desses princípios.

A ideia de criar um website único é radiante, mas que pode confundir equipes inteiras, incluindo gestores. É bom que você leia muito sobre este assunto para tomar boas decisões futuramente. Produzir um website preparado para qualquer tipo de tela é um desafio em vários momentos. Manter uma performance decente, com código legível e organizado, compatível com todos os browsers é um desafio para vida toda.

Mobile First

Mobile First é um termo muito popularizado pelo Luke Wroblewski, um dos gurus quando se trata de design para mobiles. A ideia do Mobile First é muito simples: foque-se primeiro em dispositivos com telas pequenas, depois vá adaptando seu site ou produto para telas grandes.

Quando essa loucura de web mobile iniciou, não havia muito o que os desenvolvedores pudessem fazer. Os mobiles mais usados eram features phones, lerdos, com um sistema operacional horrível e browsers péssimos. Era a época do Windows Mobile. Era a época do Symbian (esse era top). Se você for das antigas, já deve ter usado um PalmOS ou um PowerPC com orgulho. Era uma época interessante porque os aparelhos móveis eram novidade. Era difícil encontrar alguém na rua com um celular, ainda mais com um celular que se conectava com a internet.

Esse era um mercado bagunçado e muito novo. E como em todo mercado novo, não havia boas práticas em que pudéssemos nos apoiar para desenvolver websites para dispositivos móveis. Depois que o iPhone mudou todo o mercado, nós passamos a desenvolver websites mobile da mesma maneira que desenvolvíamos para desktop. Claro, com a ajuda das Media Queries, fazer versões mobile ficou muito mais fácil. Mas mesmo assim a metodologia ainda era a mesma: fazíamos um website se focando em desktops, depois adaptávamos para Mobile. Isso incluía colocar `display:`

`none;` em tudo o que você não queria que aparecesse no celular.

O ponto era que invariavelmente os celulares precisam de muita atenção. Eles são aparelhos com menos poder de processamento que os desktops/notebooks, tem um contexto mais restritivo por causa do tamanho da tela e as formas de uso são totalmente diferentes do que os usuários estão acostumados. Simplesmente mudar o tamanho das imagens, os tamanhos de texto e o posicionamento dos elementos não é suficiente. É praticamente o trabalho de fazer dois websites… não fazia sentido. Foi aí que o Luke Wroblewski cunhou o termo Mobile First.

A ideia do Mobile First é que você se foque primeiro em telas pequenas, depois nas telas grandes. Isso quer dizer que você inicia todo o planejamento, design e código do projeto com foco primeiro em telas pequenas. Isso lhe trará uma série de vantagens posteriormente, quando for o momento de desenvolver a versão desktop. Quando você pensa primeiro em telas pequenas, você precisa se preocupar com uma série de restrições. Por exemplo, performance.

Mesmo que os celulares de hoje em dia sejam poderosos e rápidos, ainda não dá para comparar com o poder de processamento de um desktop. Portanto, ao fazer um website pensando primeiro na versão mobile, você será obrigado a pensar na performance do site. Essa não era uma preocupação tão constante antes, mas agora você precisa entregar um website rápido por causa do contexto dos mobiles, onde a conexão é péssima na maioria das vezes. Note que se você já pensou na melhor performance possível, quando você fizer a versão desktop do website, performance já não será uma preocupação constante.

Isso acontecerá com uma série de outras restrições, como de fluxo de navegação, quantidade de textos, agrupamento informação e adaptação de componentes. Há algumas dicas importantes que precisam ser levadas a sério ao fazer um site mobile:

- *Se o fluxo é longo, diminua*. Lembre-se que o usuário está em um dispositivo móvel e provavelmente ele estará em um contexto estranho, como dirigindo, andando na rua ou no ônibus cheio. O fluxo de navegação não pode ser complexo nem comprido. Tente fazer um fluxo mais brando e simples, tanto para mobile quanto para desktop.
- *Mantenha comportamentos similares*. Tente sempre manter o mesmo comportamento de interações entre os dispositivos. Se você abre uma modal no Desktop, tente abrir essa mesma modal no Mobile. Claro, na medida do possível, existem cenários em que não é possível manter o mesmo comportamento em dispositivos tão diferentes, mesmo assim, sempre priorize manter comportamentos similares entre os aparelhos.
- *Imagens*. Aos poucos o problema das imagens vai se resolvendo. O Google tem se esforçado com o padrão WebP e o W3C oficializou a tag `picture`. Até que essas tecnologias estejam usáveis, não perca tempo fazendo vários tipos de imagens.
- *Não abuse dos elementos adaptados*. Há momentos em que os elementos que você tem no desktop não poderão ser usados no Mobile. Isso é normal. Você está em um ambiente mais restritivo e por isso precisa adaptar esse elemento dinamicamente via back-end ou JavaScript. Não abuse dessas artimanhas. Manter dois códigos para adaptar um mesmo elemento em telas de tamanhos diferentes pode custar muito trabalho.
- *Design e estruturas similares*. Se você trabalha com um design muito similar entre os

dispositivos, ou seja, se você usa os mesmos elementos e componentes entre telas grandes e pequenas, o usuário não vai precisar aprender a usar seu site mobile. A experiência de uso será a mesma entre desktop e celular e isso é bom.

Um pouco sobre Responsive Web Design

Você já deve ter ouvido falar sobre Responsive Web Design. Você pode ler sobre isso aqui (http://bit.ly/pcrwxY) e aqui (http://bit.ly/mSCRSD).

Até pouco tempo atrás navegávamos na web praticamente apenas via desktops. Podíamos acessar mal e porcamente a internet pelos celulares ou e por alguns aparelhos aleatórios, como o seu console, mas nenhum destes meios permitia a mesma facilidade de um desktop. Felizmente, o cenário mudou muito. Os dispositivos móveis tomaram conta. Até os celulares mais básicos contam com um browser minimamente capaz de suportar grande parte dos padrões web. Há também os tablets, que demoraram para aparecer, mas agora trazem flexibilidade para usuários que querem algo mais prático que um notebook e mais confortável que um smartphone. E estamos falando aqui de dois aparelhos que se tornaram populares há pouco tempo.

Quando não restringimos os cenários a aparelhos temos um horizonte muito maior e mais frutífero. Entenda que a informação publicada na web pode e é totalmente reutilizada a qualquer momento. O Google faz isso com seu robô todos os dias, a todo instante. O robô do Google ou o de qualquer outro sistema de busca é o que eu costumo de chamar de *meio de acesso*. O leitor de tela do usuário deficiente visual também é um meio de acesso. O leitor de RSS utilizado pelo seu celular, por mais simples que seja, é um meio de acesso. Podemos dizer que qualquer dispositivo que o usuário utilize para consumir informação na web é um meio de acesso.

O que é Responsive Web Design?

Responsive Web Design é acima de tudo um conceito. Nós nos responsabilizamos por apresentar a informação de forma acessível e confortável para diversos meios de acesso. Muitos websites restringem o conceito a aparelhos com telas de diversos tamanhos, mas o conceito é muito mais abrangente. Mesmo assim, vou restringir os primeiros exemplos a dispositivos que tenham telas e que estão mais presentes atualmente. Não vou me estender muito a meios de acesso como leitores de tela, robôs de busca ou outros dispositivos.

O primeiro passo parar tornar a web mais acessível para qualquer meio de acesso é fazer com que os browsers formatem de maneira mais eficiente nossos layouts. Isso é possível fazendo com que ele renderize um bloco de CSS de acordo com as características dos dispositivos. Isso é feito com as Media Types e as Media Queries que você vai ler mais para a frente.

O segundo passo é fazer com que o layout seja amigável. Para isso, você precisa entender os dilemas dos seus layouts e resolvê-los sem que o design mude nos dispositivos da água para o vinho, mantendo as mesmas características e histórias de uso. As premissas que envolvem o Responsive Web Design são as seguintes:

1. Grid fluído
2. Imagens Flexíveis
3. Media Queries

7.2 Media Queries

Eu não posso explicar sobre Media Queries sem falar antes das Media Types. As Media Types foram a primeira versão de um esforço para direcionar a formatação CSS para determinados tipos de meios de acesso.

O HTML foi criado para ser portável, ou seja, ele deve ser lido e interpretado por qualquer tipo de dispositivo. Cada um exibe HTML de uma determinada maneira. Logo, a forma como você formata o layout precisa ser diferente para cada dispositivo. Por exemplo, se você visita um site por um desktop, a experiência será totalmente diferente caso você visite o mesmo site por um dispositivo móvel. São diferentes, com formas totalmente distintas de navegação e uso.

Mas o exemplo anterior é muito comum. Existem outros cenários que precisamos prever para controlar a formatação do site, por exemplo, quando o usuário imprime sua página. Quando alguém imprime a página de um artigo no site do UOL, Terra ou qualquer site de conteúdo, vários elementos se tornam inúteis, começando pelo menu, barra lateral, rodapé etc. O texto poderia ser mais bem formatado para que a leitura em papel fique mais confortável. Essa diferença entre dispositivos é controlada pelas `Media Types`. Veja uma lista do que pode ser controlado via CSS:

- *all*: este valor é usado para que o código CSS seja aplicado para todos os dispositivos.
- *braille* para dispositivos táteis.
- *embossed* para dispositivos que imprimem em Braille.
- *handheld* para dispositivos de mão, celulares e outros dispositivos deste perfil. Normalmente com telas pequenas e banda limitada.
- *print* para impressão em papel.
- *projection* para apresentações, como PowerPoint. Este valor foi inventado pelo pessoal da Opera. **muito** útil.
- *screen* para dispositivos com telas coloridas e alta resolução.
- *speech* para sintetizadores de voz. O CSS 2 tem uma especificação de CSS chamada Aural (http://www.w3.org/TR/CSS2/aural.html) com que podemos “formatar” a voz de leitores de tela e outros sintetizadores.
- *tty* para dispositivos que utilizam uma grade fixa para exibição de caracteres, como teletypes, terminais, dispositivos portáteis com display limitado.
- *tv* para dispositivos como televisores, ou seja, com baixa resolução, com boa quantidade de cores e scroll limitado.

A aplicação é muito simples: basta adicionar a linha comum de `link` para seu CSS, inserindo um atributo `media` e adicionando o valor desejado:

```
1 <link rel="stylesheet" href="estilo.css" media="handheld">
```

Note que nesse exemplo estou chamando um arquivo CSS, que será destinado para funcionar em dispositivos de media *HANDHELD*: aparelhos móveis, celulares com tela pequena e aparelhos similares (já usou PalmTop?). Logo, esse CSS não será aplicado quando o usuário visitar o site utilizando um desktop. Para tanto, teríamos que utilizar media *SCREEN*:

```
1 <link rel="stylesheet" href="estilo.css" media="screen">
```

O problema

Você já notou que todo dia surgem novos dispositivos, com diversos tamanhos e hardwares parecidos com os desktops? Qualquer celular meia-boca hoje tem a configuração mais parruda que qualquer computador antigo. Principalmente a configuração da tela, à qual os fabricantes têm dado mais atenção nos últimos anos. Logo, não há motivo para prepararmos um layout e um CSS com media type *HANDHELD* para o iPhone, já que ele não se encaixa nessa categoria. Entretanto, o iPhone também não é nem de longe um desktop. Aí existe o problema: a media type screen se encaixaria para direcionarmos a formatação para o iPhone e outros smartphones modernos, mas a especificação é clara quando diz que a media type screen é para desktops e computadores. Como fazer agora?

A solução: Media Queries

As Media Queries definem condições para que o CSS seja utilizado em cenários específicos. Se essas condições forem aprovadas, ou seja, se o dispositivo se adequar a todas as condições estabelecidas na sua Media Querie, o CSS será aplicado. Veja um exemplo:

```
1 <link href="estilo.css" media="screen and (max-width: 480px)">
```

Especificamos que o arquivo `estilo.css` será aplicado em dispositivos que se enquadram na condição de `screen` (ou seja, que têm uma tela com alta capacidade de cores) e com uma largura máxima de 480px.

Há uma lista de características que você pode utilizar aqui para selecionar os dispositivos que você quiser:

- width
- height
- device-width
- device-height
- orientation
- aspect-ratio
- device-aspect-ratio
- color
- color-index

- monochrome
- resolution
- scan
- grid

Geralmente, usamos as Media Queries dentro do código CSS. Você linka seu CSS no `HEAD` do documento assim:

```
1 <link rel="stylesheet" href="estilo.css">
```

Dentro do código CSS, você vai separar os famosos *breakpoints*, que são as condições de tamanho das telas dos dispositivos. Esse tamanho vai definir quando cada bloco de CSS será utilizado. Veja o código a seguir:

```
 1 /*
 2  Código que será herdado por qualquer dispositivo.
 3  Se você já conhecer a ideia do Mobile First,
 4  esse primeiro código será destinado para mobiles.
 5 */
 6 a {color: blue;}
 7
 8 /*
 9  Pra dispositivos que têm uma largura mínima de 768 pixels.
10  Tablets, por exemplo.
11 */
12 @media (min-width: 768px) {
13   a {color: yellow;}
14 }
15
16 /*
17  Com uma largura mínima de 992 pixels. Monitores por exemplo.
18 */
19 @media (min-width: 992px) {
20   a {color: green;}
21 }
22
23 /*
24  Dispositivos com largura mínima de 1200 pixels. Por exemplo
25  grandes monitores e TVs.
26 */
27 @media (min-width: 1200px) {
28   a {color: black;}
29 }
```

E assim você vai escrevendo seu CSS e manipulando a formatação do layout de acordo com o dispositivo desejado.

7.3 Metatag Viewport

O *viewport* é a área onde seu website aparece. É aquela área branca onde o website é montado pelo browser. É a área na qual você se preocupa se o layout vai ou não caber na hora da criação. O tamanho do viewport depende muito da resolução, tamanho do monitor e dispositivo utilizado. Em máquinas desktop, nós não precisamos nos preocupar muito, já estamos acostumados com um determinado tamanho de tela e resolução média utilizada pelos usuários, que hoje gira em torno de no mínimo 1024 de largura. Mas quando começamos a variar muito o tamanho das telas, a largura do viewport começa a ser uma preocupação porque afeta diretamente a forma como o usuário utiliza seu website.

Hoje existe uma gama muito grande de aparelhos com telas de tamanhos variados. Comece a pensar pelos netbooks e depois vá diminuindo até chegar em um smartphone popular como o iPhone etc. A variação dos tamanhos é muito grande. A largura da tela do iPhone na posição retrato é de 320px. Mas sua resolução padrão é 980px. Por isso, tenha em mente que *tamanho de tela* é bem diferente de *resolução*. Isso quer dizer que, se você fizer um HTML simples, colocar uma imagem de 980px de largura e visualizar no iPhone não existirá barra de rolagem. Obviamente a imagem ficará **miniaturizada**, como mostrado a seguir:
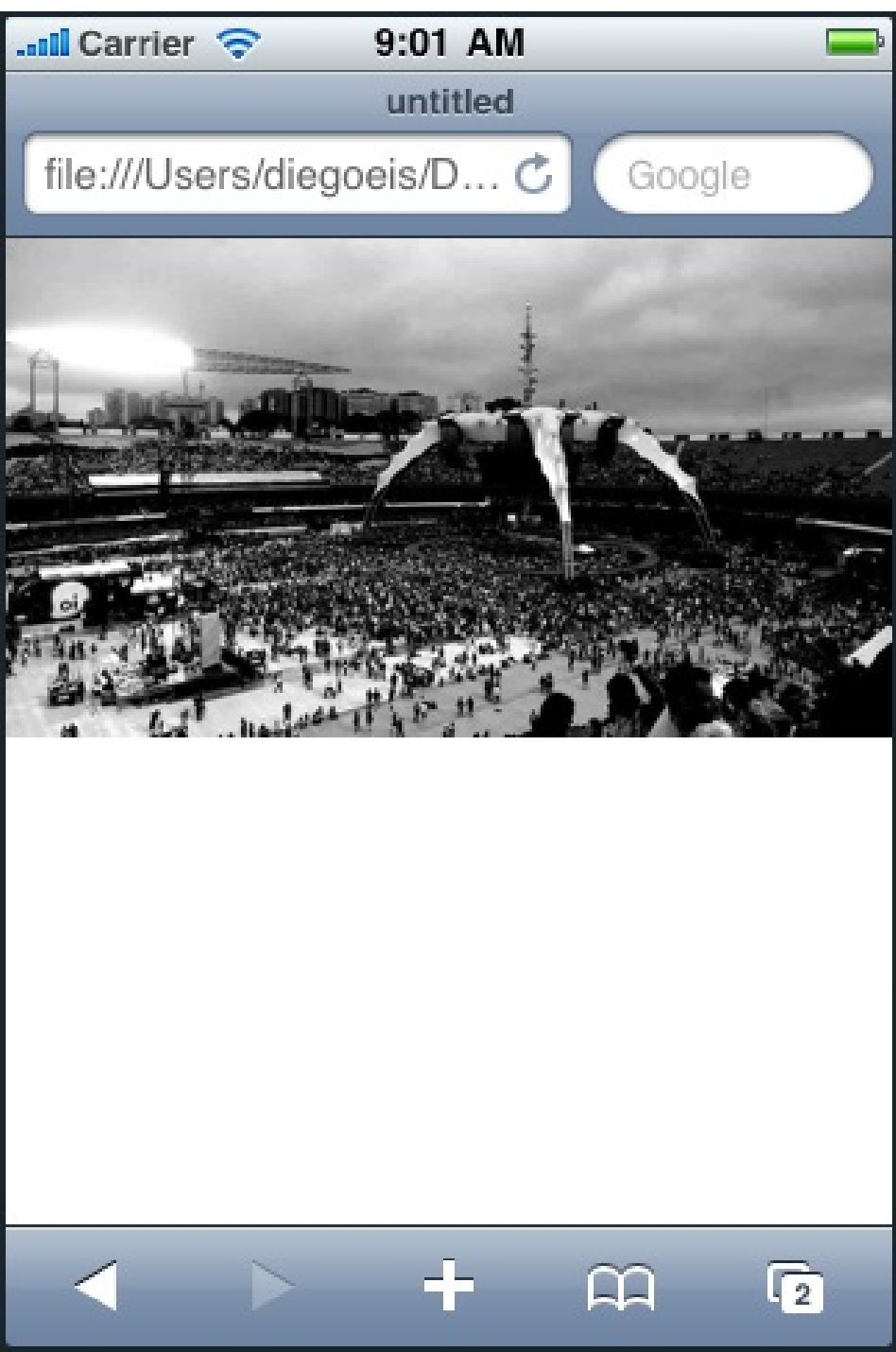
Fig. 7.1: Sem metatag viewport, o celular redimensiona a imagem.

Isso é interessante porque possibilita a boa visualização de websites que não estão preparados para mobiles. Vemos o site inteiro miniaturizado, com possibilidade de fazer zoom em qualquer parte do layout.

Manipulando a meta tag Viewport

Mesmo assim, os smartphones têm telas pequenas que podem dificultar a leitura se fizermos um sistema planejado para grandes resoluções. Por isso, é interessante que possamos customizar o viewport para que ele se adeque à realidade desses dispositivos. É aí que entra a *metatag viewport*.

Com essa metatag, vamos preparar o viewport para a resolução que o browser deve renderizar. Dessa forma, podemos preparar websites com resoluções personalizadas para smartphones e outros aparelhos. A sintaxe é muito simples e deve ser colocada, como sempre, na tag `head`. Veja o código inicial:

```
1 <meta name="viewport" content="width=device-width">
```

Os valores de `content` são os seguintes:

- `width` define uma largura para o viewport. Os valores podem ser em PX ou `device-width`, que determina automaticamente um valor igual à largura da tela do dispositivo.
- `height` define uma altura para o viewport. Os valores podem ser em PX ou `device-height`, que determina automaticamente um valor igual à altura da tela do dispositivo.
- `initial-scale` define a escala inicial do viewport.
- `user-scalable` habilita a possibilidade de o usuário fazer zoom na tela. É ativado quando o usuário bate duas vezes com o dedo em um lugar da tela ou faz o movimento de *pinch*.

Para facilitar o uso, definimos que o viewport irá sempre renderizar o site com a largura default do dispositivo usado pelo usuário com o valor `device-width` na propriedade `width`. Existe também o valor `device-height`, que é usada na propriedade `height`, quando seu layout mudará de posição caso a altura da viewport seja modificada (como quando o usuário usar um aparelho com uma altura maior do que o normal).

```
1 <meta name="viewport" content="width=device-width">
```

Este é um assunto que merece muito estudo e por isso seguem alguns links para que você se aprofunde mais. É muito necessário para a produção de layouts para celulares.

- Lista de tamanhos de viewports dos principais dispositivos (http://i-skool.co.uk/mobile-development/web-design-for-mobiles-and-tablets-viewport-sizes/)
- Artigo da Mozilla Developer Network (https://developer.mozilla.org/pt-BR/docs/Mozilla/Mobile/Viewport_meta_tag)
- Manipulando a metatag Viewport (http://tableless.com.br/manipulando-metatag-viewport/)
- A regra @viewport (http://tableless.com.br/regra-viewport/)

7.4 Unidades em REM

Aqui no Brasil, é muito comum usarmos pixels para definição de tamanho de textos. Lá fora a galera gosta muito de usar EM. Mas qual a diferença?

Pixels

Unidade em pixels é mais velho que andar para trás. Você utiliza pixels para definir a largura de um

elemento, o tamanho do texto, a espessura da borda e outras coisas.

Unidade em Pixels é utilizado porque é a medida mais exata que você pode encontrar. Por não ser uma medida variável, os pixels são fáceis de controlar. Fáceis de calcular. Você abre seu Photoshop, mede e passa os valores para CSS facilmente. É tudo muito eficiente. É por isso que a maioria dos devs sempre preferiu utilizar pixels nos projetos.

Mas, antigamente, definir unidades de texto em pixels trazia uma desvantagem herdada do Internet Explorer: quando o usuário tentava mudar o tamanho do texto pelo browser, por algum motivo bizarro o IE não aumentava a font porque ela estava definida em pixels. Um problema sério de acessibilidade. É por isso que muitos devs, durante um tempo, definiram o tamanho do texto utilizando % (porcentagem). O problema é que trabalhar com porcentagem é algo muito instável. Havia diferenças de tamanhos de textos entre os browsers e por causa disso o layout nunca ficava igual.

Unidades em EM

EM é uma unidade de medida tipográfica e seu nome está relacionado à letra `M`. O tamanho base dessa unidade deriva da *largura da letra M em maiúscula*. Dizem que *1 em* equivale aproximadamente a *16 pontos*.

Não sou eu que estou falando isso, é a Wikipedia (http://tableless.com.br/unidade-pixels-em-rem/“http://en.wikipedia.org/wiki/Em_(typography)”). ;-)

O problema de utilizar fonts em EM é que elas, como as porcentagens, são variáveis. Diferentemente da utilização de pixels, temos que fazer um pouco de matemática para planejar nossas unidades no projeto. Não é nada de outro mundo, então pare de preguiça.

A fórmula é fácil de entender. Vou manter os termos em inglês para você entender melhor quando for procurar mais informações sobre este assunto:

target ÷ context = result

Um exemplo: imagine um título com o `font-size` de 20px.

```
1 h1 {
2   font: 20px verdana, arial, tahoma, sans-serif;
3 }
```

O `target` (que é o elemento que queremos modificar) é *20px*. Nesse caso, o `BODY` é o pai do nosso H1, portanto ele é nosso `context`, que como já dissemos tem o valor padrão de *16px*.

20 ÷ 16 = 1.25.

```
1 h1 {
```

```
2    font: 1.25em verdana, arial, tahoma, sans-serif;
3 }
```

Se este H1 estiver dentro de outro elemento, tipo um div, o valor de `context` agora é o tamanho da font do div. Tenha como exemplo este HTML:

```
1 <div>
2     <h1>Um título bacana</h1>
3     <p>Um texto grande e bacana para fazermos parágrafos.</p>
4 </div>
```

O CSS com as fonts em PX:

```
 1 div {
 2     font: 30px verdana, arial, tahoma, sans-serif;
 3 }
 4 
 5 h1 {
 6     font-size: 20px;
 7 }
 8 
 9 p {
10     font-size: 12px;
11 }
```

Primeiro parágrafo: *12px* (target) ÷ *30px* (context [div]) = *0.4em*

Título: *20px* (target) ÷ *30px* (context [div]) = *0.67em*

Div: *30px* (target) ÷ *16px* (context [body]) = *1.88em*

```
 1 body {font: 100% verdana, arial, tahoma, sans-serif;}
 2 
 3 div {
 4     font-size: 1.88em;
 5 }
 6 
 7 h1 {
 8     font-size: 0.67em;
 9 }
10 
11 p {
12     font-size: 0.4em;
13 }
```

Em vez de você mudar as fonts de cada elemento, você pode simplesmente mudar o valor da font do target, ou seja, do body para 120% ou para 80%, isso fará com que todas as fonts do site aumentem ou diminuem proporcionalmente em relação à porcentagem do body.

Infelizmente isso não é para todos os browsers: Firefox 3.6+, Chrome, Safari 5 e IE9 suportam a unidade REM. Mas e os que não suportam? Bom, use a font em PX. Cá entre nós, dessa lista de browsers só faltou o IE8, já que o IE7 e 6 já foram embora. Se você não suportar o IE8 em seus projetos, faça um fallback como o exemplo a seguir. Você vai escrever um pouco mais de código, mas vai garantir a compatibilidade do IE8.

```
 1 body {font: 100% verdana, arial, tahoma, sans-serif;}
 2 
 3 div {
 4     font-size: 30px;
 5     font-size: 1.88rem;
 6 }
 7 
 8 h1 {
 9     font-size: 20px;
10     font-size: 1.25rem;
11 }
12 
13 p {
14     font-size: 12px;
15     font-size: 0.75rem;
16 }
```

Enquanto o REM/EM são mais flexíveis e nos dá a vantagem da acessibilidade, por outro lado envolve um pouco de matemática. Mesmo assim, utilizar REM tem sido uma ótima prática e não tem trazido muitos problemas, na minha experiência. Como uso SASS, faço uma Mixin para calcular REM e transformar em PX.

7.5 Imagens

Trabalhar com imagens em layouts responsivos sempre foi algo limitado. Antes de aparecer a tag `picture` e o `srcset`, podíamos no máximo manipular o tamanho da imagem com `max-width` ou usar plugin malucos para procurarem em nossos servidores uma imagem com mais ou menos qualidade dependendo do nosso dispositivos. Tudo era uma grande gambiarra. Mas agora a coisa mudou. Adiante, você conhecerá duas tecnologias que nos ajudam a servir imagens de forma inteligente para sites responsivos.

Existe um grupo chamado *Responsive Images Community Group* (http://responsiveimages.org/) formado por desenvolvedores para discutir e desenvolver soluções para o problema das imagens

responsivas, com uma solução client-side, padronizada e baseada nas capacidades do dispositivo usado pelo usuário. Isso melhora o gasto excessivo de banda e otimiza o uso de imagens para os diferentes tipos de tela e também para impressão.

Nesse grupo são discutidas uma série de opções, mas as soluções que já saíram do papel e estão sendo suportadas por alguns browsers são a tag `<picture>` e o atributo `srcset`.

Casos de uso

Existem diversos casos em que precisamos mostrar a mesma imagem, mas com resoluções diferentes. A seguir, descrevi alguns destes casos. Uma lista mais detalhada e completa pode ser vista da documentação do W3C:

Baseado em resolução — Geralmente, queremos mostrar a mesma imagem em múltiplas resoluções. Em dispositivos com uma resolução ótima, podemos servir imagens com mais qualidade do que para aparelhos com resoluções piores.
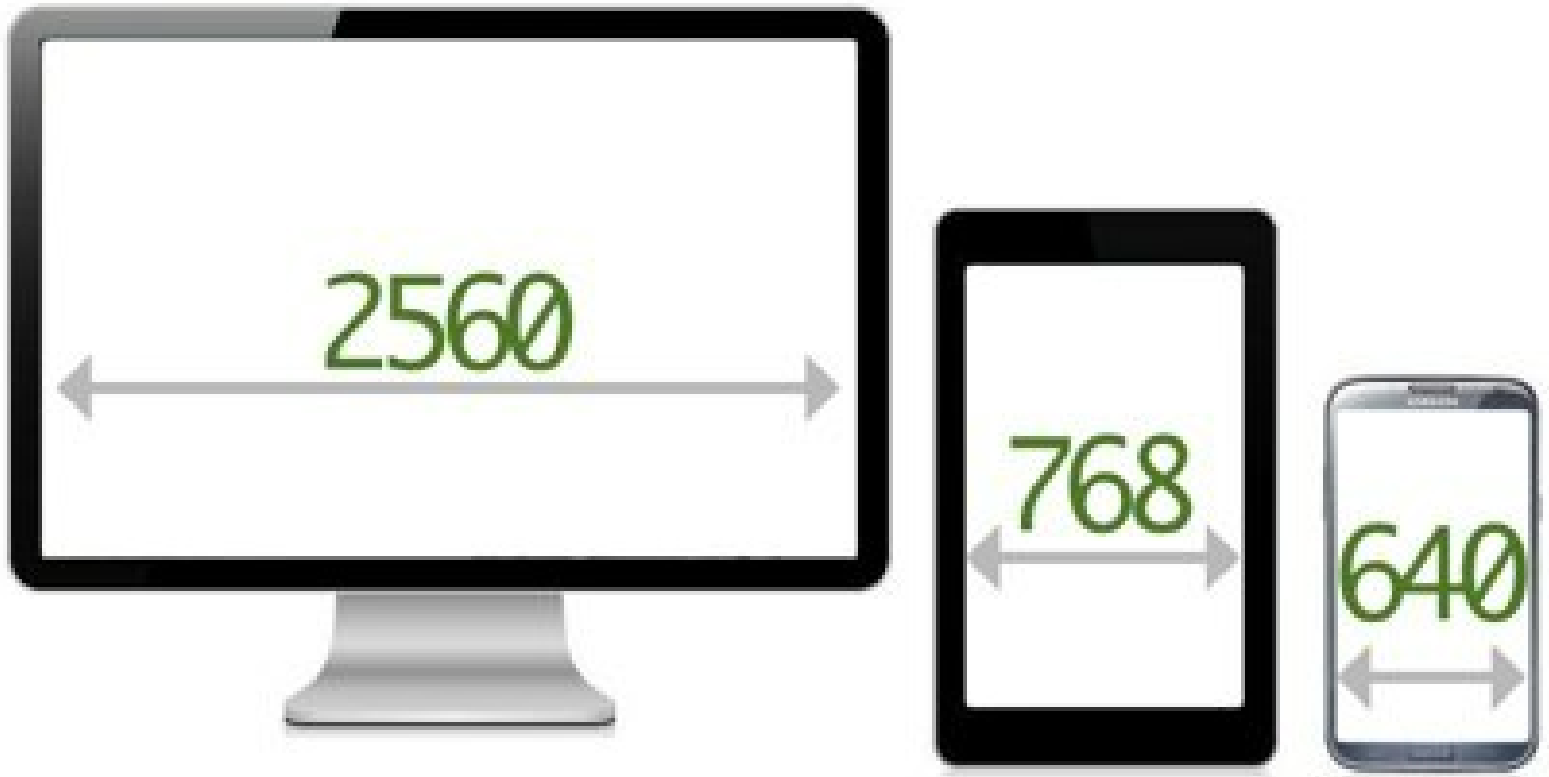
Fig. 7.2: Vários dispositivos, várias resoluções.

Baseado no viewport — Podemos também servir imagens se baseado no viewport do usuário, detectando se ele está usando um aparelho com tela pequena ou com tela grande. Por exemplo: em sites que têm aquelas imagens gigantes cobrindo o `header`, ao estilo do Medium. Aquela imagem grande fica muito bem em aparelhos com viewport grande, mas para aparelhos pequenos, a imagem é muito grande e pesada pra ser carregada.
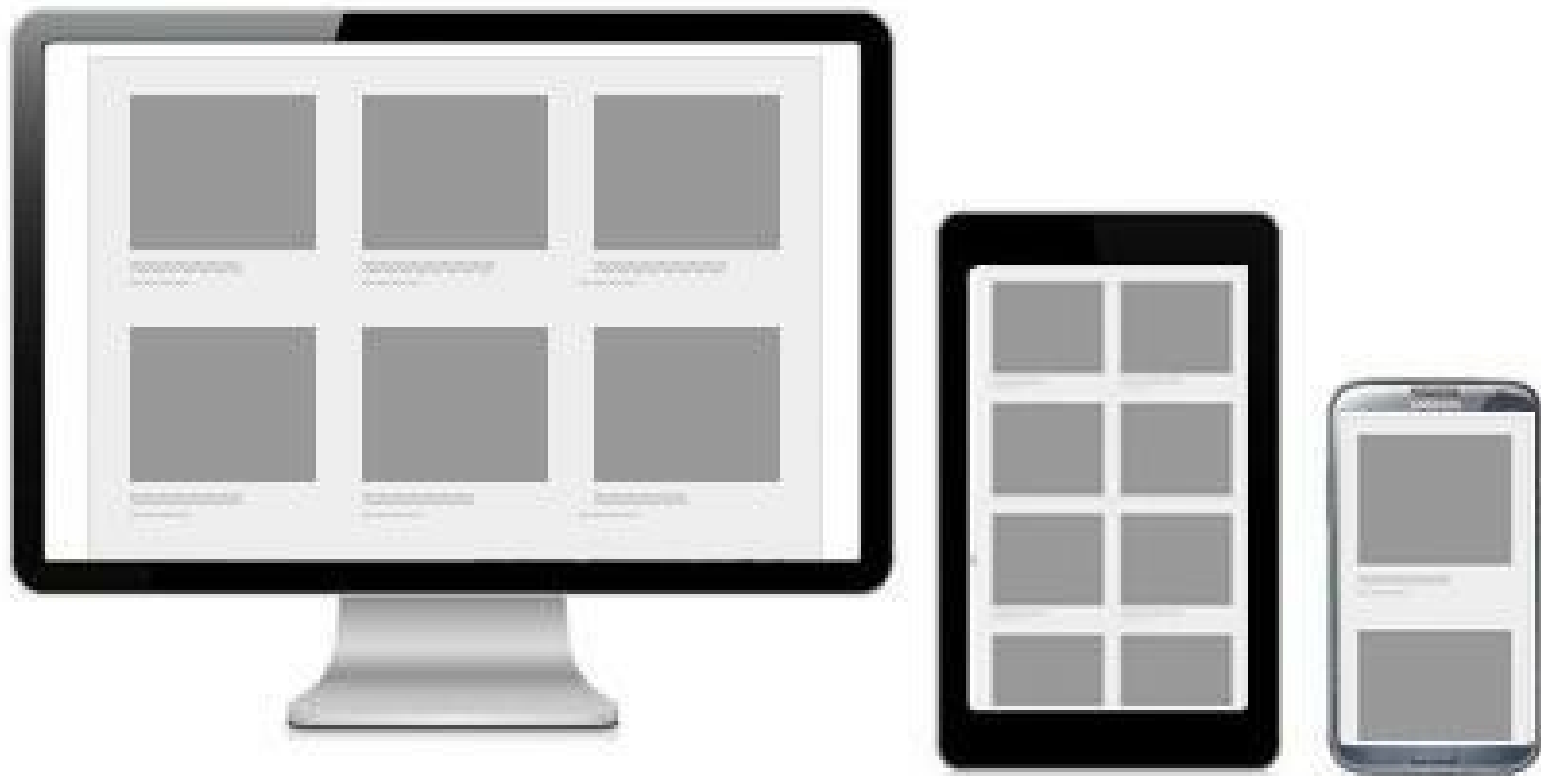
Fig. 7.3: Lembre-se: viewport é a área disponível onde o site irá aparecer.

Densidade de pixels — Há também o cenário onde queremos mostrar imagens melhores para dispositivos que têm uma densidade de pixels maior, evitando servir para imagens com baixa qualidade para esses dispositivos. Quanto maior a densidade de pixels do aparelho, maior a quantidade de pixels a imagem precisa ter.
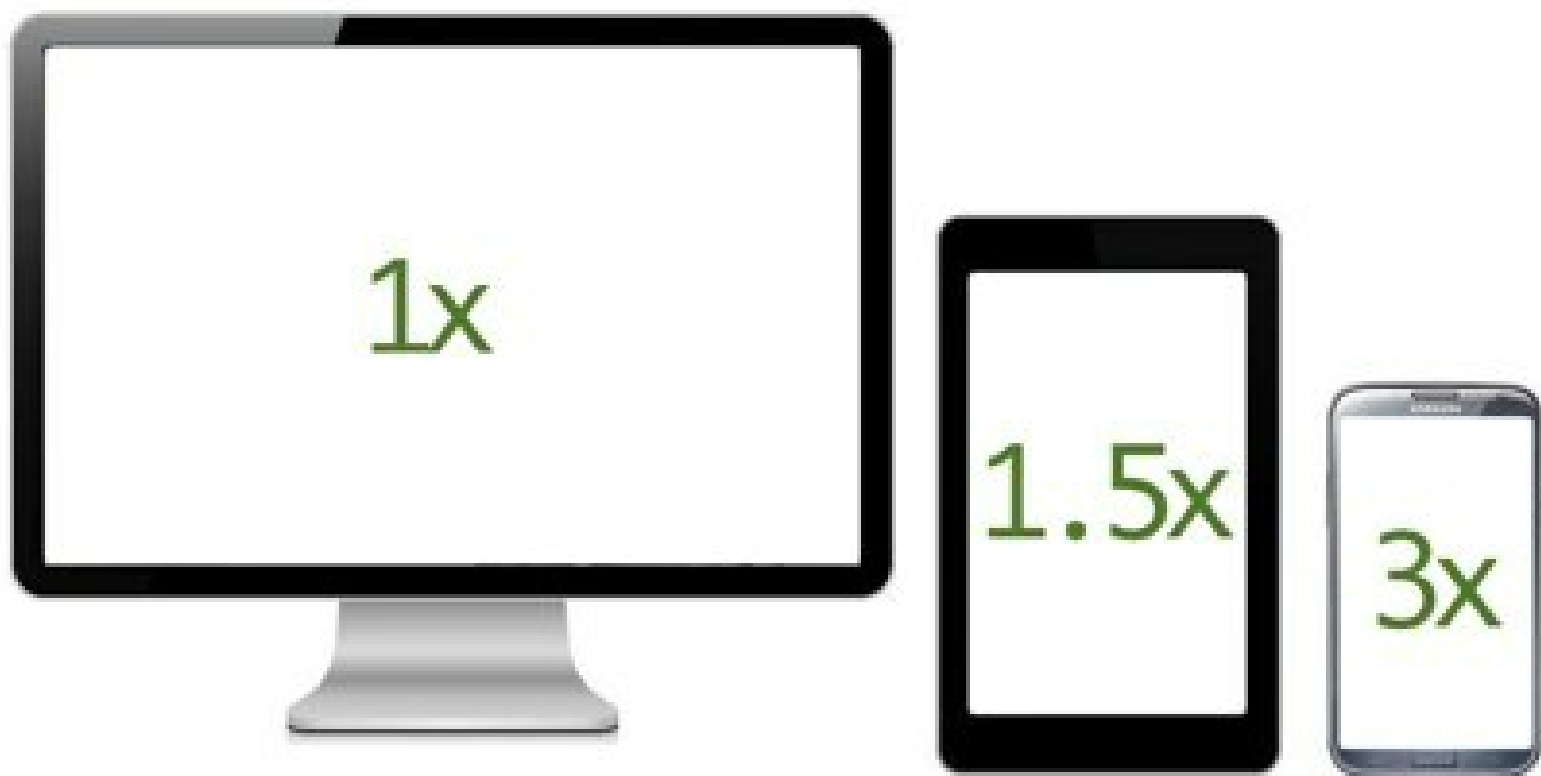
Fig. 7.4: Os dispositivos têm densidade de pixels diferente, independente do seu tamanho.

Direção de arte — É um cenário onde podemos mostrar a mesma imagem, com recortes diferentes.

Fig. 7.5: Mesma imagem, recortes diferentes.

Link da documentação com os casos de uso: http://usecases.responsiveimages.org/#use-cases

O atributo SRCSET

O atributo `srcset` permite especificar uma lista de fontes para uma chamada de imagem, entregando a imagem correta de acordo com a densidade de pixels da tela do usuário. O código fica assim:

```
1 <img src="imagem-baixa-resolucao.jpg"
2   srcset="imagem-alta-resolucao.jpg 2x">
```

Se você quiser contemplar telas com uma densidade de pixels maior, basta acrescentar mais opções:

```
1 <img src="imagem-baixa-resolucao.jpg"
```

```
2 srcset="imagem-alta-resolucao.jpg 2x, imagem-maior-ainda.jpg
3x">
```

Muito simples, não é? O browser simplesmente mostra a imagem de baixa resolução como padrão e se o usuário estiver usando uma tela de alta resolução, o browser se encarrega de mostrar a imagem em alta. Esse é um truque que você pode usar agora. Se o browser não reconhecer o atributo `srcset`, ele simplesmente vai ignorar e seu site não vai quebrar.

Por curiosidade: rolou uma grande discussão sobre a semântica do atributo `srcset`. Uns dizem que um atributo serve para que o elemento herde algumas características. Isso quer dizer que usar um atributo como fallback para servir outras fontes de arquivos é errado. Mas convenhamos, a sintaxe é muito simples. Você pode ver o contexto dessa discussão sobre a sintaxe nesse artigo do A List Apart: http://alistapart.com/article/responsive-images-and-web-standards-at-the-turning-point

É aí que entra a tag `<picture>`.

O elemento picture

O elemento `picture` é um padrão de marcação que permite declarar uma série de fontes alternativas para uma imagem. Usando Media Queries, temos controle de quando mostrar cada uma dessas imagens para o usuário.

A sintaxe é bastante simples:

```
1 <picture>
2     <source media="(min-width: 980px)" srcset="large.jpg">
3     <source media="(min-width: 768px)" srcset="med.jpg">
4     <img src="small.jpg" alt="The president giving an award.">
5 </picture>
```

Note ali que há um atributo `media` em cada element `source` dentro de `picture`. Ele define o breakpoint que o browser vai baixar aquela imagem específica.

É por isso que há tanta discussão quanto ao uso do elemento `picture`: teoricamente ele resolve apenas o cenário de "direção de arte", que é quando precisamos servir uma mesma imagem com recortes ou tamanhos diferentes. Em outros cenários, o atributo `srcset` resolve bastante bem.

7.6 Muito mais

Existem muitos, mas muitos assuntos que poderiam ser abordados aqui, mas aí este livro se tornaria exclusivamente sobre desenvolvimento mobile. Como este não é meu intuito, prefiro deixar vários assuntos para você estudar e se divertir um pouco, ok? São materiais de pesquisa que pouparão um tempo valioso. Acho importante avisar que alguns links estão em inglês, mas nem preciso dizer que

saber inglês é requisito mínimo para você trabalhar com web, né?

- Artigo: Não use o element picture na maioria da vezes (http://blog.cloudfour.com/dont-use-picture-most-of-the-time/)
- Artigo: O cenário do Web Design Responsivo (http://tableless.com.br/o-cenario-do-web-design-responsivo/)
- Artigo: Design Responsivo por uma web Única (http://sergiolopes.org/responsive-web-design/) Artigo: Responsive Web Design — Adaptação vs Otimização (http://tableless.com.br/responsive-web-design-adaptacao-vs-otimizacao/)
- Artigo: Tudo o que você precisa saber sobre resoluções, telas retina, devicePixelRatio, DPI, CSS Pixels e etc (http://sergiolopes.org/resolucoes-dpi-pixel-ratio-retina/)
- Artigo: Textos do Jason Grigsby sobre imagens responsivas (http://blog.cloudfour.com/?s=responsive+images)
- Artigo: As muitas faces do Mobile First (http://tableless.com.br/as-muitas-faces-do-mobile-first/)
- Artigo: WebKit Has Implemented srcset, And It's A Good Thing (http://www.smashingmagazine.com/2013/08/21/webkit-implements-srcset-and-why-its-a-good-thing/)
- Artigo: "RWD Is Bad for Performance" Is Good for Performance (http://timkadlec.com/2014/07/rwd-is-bad-for-performance-is-good-for-performance/)
- Responsive Web Design Newsletter (http://responsivedesignweekly.com/)
- Slides: Como suportar telas de alta resolução eficientemente na web (http://sergiolopes.org/palestra-retina-web/#slide-capa)
- Slides: *Este é essencial*: For a Future-Friendly Web (http://www.slideshare.net/bradfrostweb/for-a-futurefriendly-web-webvisions-chicago-2012)
- Slides: O blá blá blá do Responsive Web Design ou Os cuidados do Web mobile (http://pt.slideshare.net/diegoeis/o-bla-bla-bla-do-responsive)
- Slides: Responsive Design Vs Separate Mobile Sites: Presidential Smackdown Edition (http://bit.ly/1E5rSnW)
- Slides: Multi-device Web Design (http://www.lukew.com/presos/preso.asp?31)
- Slides: Mobile input (http://www.lukew.com/presos/preso.asp?23)
- Slides: Mobile First (http://www.lukew.com/presos/preso.asp?26)
- Slides: Design for touch (http://www.lukew.com/presos/preso.asp?33)
- Livro: Mobile First (http://www.lukew.com/resources/mobile_first.asp)
- Livro: Web Design Responsivo: Páginas adaptáveis para todos os dispositivos (http://www.casadocodigo.com.br/products/livro-web-design-responsivo)
- Livro: A Web Mobile: Programe para um mundo de muitos dispositivos (http://www.casadocodigo.com.br/products/livro-web-mobile)
- Diretório com vários links escolhidos pelo Sérgio Lopes (http://sergiolopes.org/diretorio-design-responsivo/)

Performance

Geralmente, 80% da performance de um website ou um produto de internet estão concentrados no front-end. Mesmo assim, performance é um daqueles assuntos que todo mundo sabe que é importante, mas que grande parte das equipes, por diversos motivos, não tratam como uma feature do projeto. A grande maioria só vai pensar em melhorar a velocidade do projeto apenas no final do desenvolvimento, quando seu HTML, CSS e JS já foram feitos ou até pior, quando o produto já está publicado e funcionando. O correto é que isso seja uma preocupação durante todo o processo de desenvolvimento.

O ideal é que os devs comecem a pensar em performance desde o planejamento do projeto, junto com o wireframe. Existem muitas decisões de UX, por exemplo, que podem influenciar bastante a agilidade e o carregamento do produto. Este é um dos motivos pelo qual pensamos sempre em Mobile First.

Este é um assunto bastante extenso, quero apenas abordar umas poucas coisas que julgo serem mais importantes para você, que é iniciante. Guarde também este termo para uma pesquisa posterior: *Performance as a feature*. Essa é uma expressão que tem sido usada para demonstrar a preocupação com performance como se fosse uma feature do projeto.

8.1 O processo

O browser segue alguns passos importantes antes de mostrar sua página na tela do usuário. É importante você conhecer o que acontece por baixo dos panos para entender melhor tunar seu produto. Os passos são:

1. O browser monta o DOM (*Document Objecto Model*) do documento. O DOM é uma representação do que será seu HTML. Ele é essencial para o browser entender a relação que os elementos têm entre si. Mas lembre-se: o DOM não é o HTML. O HTML é o resultado final.
2. O browser monta o CSSOM (*CSS Object Model*), que é uma representação, como o DOM, que guarda as regras de formatação dos elementos do HTML.
3. O browser junta as duas árvores, o DOM e o CSSOM, em uma única árvore chamada *RenderTree*. Na RenderTree o browser monta uma árvore baseada nos elementos que serão visíveis para o usuário. Isso quer dizer que elementos que têm `display: none` não vão constar na RenderTree.
4. Depois juntar as duas árvores, o browser calcula a geometria dos elementos, colocando largura e altura. Feito isso ele posiciona o elemento na tela. Essa fase se chama Layout.
5. Sabendo do seu tamanho e sua posição, o browser entra da fase chamada Paint. Nessa fase ele irá pintar os elementos e seus filhos na tela, renderizando cores de background, imagens de background, borda, drop-shadow e outras propriedades.

Existe muita ciência nesse assunto que poderá ajudá-lo a entregar um produto mais rápido e ágil. O Chrome DevTools pode contribuir bastante no monitoramento e análise desse processo, fornecendo dados importantes para você entender os gargalos.

8.2 Saiba o que é Reflow e Repaint

O browser faz duas coisas ao juntar seu HTML com o CSS: ele posiciona todos os elementos no fluxo que você definiu ao implementar o layout e depois os pinta, colocando cores no background, no texto, definindo as imagens de fundo etc. Ele faz isso tudo no carregamento do CSS. Quando um browser precisa reposicionar um elemento do DOM por causa de alguma ação do usuário ou do seu JavaScript, isso se chama *Reflow*. Quando o browser precisa pintar/desenhar novamente um elemento na tela, isso se chama de *Repaint*.

Geralmente o Reflow e o Repaint são causados quando você modifica a árvore do DOM por meio do JavaScript. O Reflow, principalmente, exige bastante do computador do usuário — inclusive CPU — pois o browser precisa fazer uma série de cálculos para reposicionar o elemento no lugar correto. Quando você faz a modificação de algum elemento, não é apenas ele que é recalculado e reposicionado, mas todos os outros elementos que o envolvem também. Além disso, quando o Reflow é executado, o browser bloqueia todas as operações do usuário. Ou seja, o usuário não pode executar nenhuma tarefa como clicar ou rolar a página.

Profundidade de código

Aqui é um bom momento de falar sobre a profundidade do seu código HTML. Quando você produz um código HTML, é muito importante que você o deixe o mais simples possível. Ou seja, quanto menos profundidade existir, melhor. Um exemplo é o código a seguir:

```
1 <nav class="menu">
2   <div class="menu-inner">
3     <ul class="menu-options">
4       <li><a href="" title="">...</a></li>
5     </ul>
6   </div>
7 </nav>
```

Seria melhor se fosse assim:

```
1 <nav class="menu">
2   <ul class="menu-options">
3     <li><a href="" title="">...</a></li>
4   </ul>
5 </nav>
```

Claro, isso depende bastante da complexidade do layout. Mas quanto menos profundo seu código, menos elementos o browser recalcula em casos de Repaint ou Reflow.

A profundidade do seu código também influencia a complexidade do seletor CSS. Quando se faz um

seletor muito “profundo”, por assim dizer, o browser sempre levará mais tempo para encontrar o elemento que será estilizado. É por isso que na maioria das vezes é uma boa prática referenciar os elementos pais com uma **classe**. Levando em consideração o código HTML que vimos agora há pouco, este seria um seletor ruim:

```
1 .menu .menu-inner .menu-options li a {…}
```

O problema desse primeiro seletor é que o browser começa a procurar **todos** os elementos `a` da página; ele apenas seleciona os `a` que estiverem dentro de um `li`. Depois disso ele segue selecionando os que estiverem dentro do `.menu-options` e assim por diante. É um longo caminho.

Você poderia fazer assim:

```
1 .menu-options a {…}
```

O browser vai procurar primeiro **todos** os elementos `a` da página, e depois vai selecionar os caras que estiverem dentro do elemento com classe `.menu-options`. Um caminho mais rápido. Você pode também inserir uma classe direto nos links. Costumo fazer isso em projetos que envolvem administrativo de produtos. Em websites médios/pequenos, uso a abordagem de colocar a classe apenas no pai.

Se você tiver curiosidade, veja como um reflow funciona neste vídeo: http://bit.ly/1xqVVF5.

Reflow e Repaint são dois assuntos bem extensos e muito, muito interessantes. Por isso, não se desespere. Agora que você já sabe que eles existem, há bastante tempo para estudar depois. Continue lendo o capítulo e no final deixei vários links com conteúdo fresco para você se esbaldar. ;-)

8.3 Onde colocar as chamadas de CSS e JS?

Parece uma pergunta besta, mas não é. A resposta é simples: chamadas de CSS ficam dentro da tag `HEAD`. As chamadas dos `JS` ficam sempre antes de fechar o `BODY`. Basicamente é isso. Claro, há exceções, mas geralmente você disponibilizará seus CSS e JS dessa forma. Vamos ver mais detalhes.

Chamando o CSS

O CSS é um recurso bloqueante. Um recurso bloqueante faz o browser parar de renderizar qualquer coisa na página enquanto esse recurso não for carregado. Isso significa que o browser para tudo quando encontra uma chamada de CSS no seu código. Se ele não fizesse isso, ele carregaria seu HTML e mostraria para o usuário o site totalmente sem estilos; poucos segundos depois, o site iria piscar e boom, o layout apareceria. O usuário teria um lixo de experiência. Isso acontece se você colocar o CSS no final do documento. Ele vai carregar seu HTML inteiro e o usuário verá o site primeiro sem nenhum estilo. Quando o browser chegar ao final do documento e encontrar a chamada

do CSS, só então vai carregá-lo e seu usuário verá o site mudar bruscamente. O Google chama esse “acontecimento” de FOUC: *Flash of Unstyled Content*.

Quando o CSS é carregado depois, isso causa Reflow e Repaint, já que o browser tem que recalcular todo o HTML já carregado e então posicioná-lo e pintá-lo novamente na tela.

Quando inserimos a chamada de CSS no `HEAD`, ele vai carregar todo o CSS e depois vai continuar o carregamento do HTML. O carregamento é progressivo, ou seja, o browser mostra os elementos na tela conforme lê seu código. Esse feedback visual é importantíssimo para a experiência do usuário, pois diminui a espera e mostra para ele um indicativo de que a página está funcionando.

Chamando o JS

O JavaScript é o cara que manipula o HTML e o CSS. Exatamente por isso ele precisa ser carregado apenas depois que seu HTML e seu CSS foram carregados pelo browser. Você deve chamar seus scripts logo antes do fechamento da tag `BODY`.

O carregamento de scripts também bloqueia os downloads paralelos. A especificação do HTTP/1.1 *sugere* que os browsers façam apenas dois requests em paralelo por *hostname*. Por exemplo: se você precisar carregar várias imagens na sua página, que estão em um mesmo domínio, o browser irá carregar apenas duas por vez. É por isso que grandes sites carregam imagens ou outros assets em domínios diferentes. Se você coloca várias imagens em vários *hostnames*, o browser vai carregar mais imagens por vez. Se terminou uma, ele busca a outra, mas sempre no limite de 2 arquivos em paralelo por domínio. Quando a chamada do script é encontrada, esse tipo de carregamento é bloqueado.

O Google mostra que existem 4 pontos que acontecem quando o script é carregado:

1. A localização do script no documento é significativa.
2. A criação do DOM é interrompida quando uma tag de script é encontrada e até que a execução do script seja concluída.
3. O JavaScript pode consultar e modificar o DOM e o CSSOM.
4. A execução de JavaScript é atrasada até que o CSSOM esteja pronto.

Se você quiser ver mais detalhes, consulte este link: http://bit.ly/1tZEYjH.

Para melhorar a velocidade de renderização da página, você pode avisar para o browser que ele pode baixar aquele script independente do carregamento do HTML. Se seu script não vai executar mudanças na árvore do DOM, esse recurso é muito útil, já que você libera o navegador para fazer outras coisas enquanto seu script é carregado.

O truque é adicionar um atributo chamado `async` à chamada do seu script no HTML. Fica assim:

```
1 <script src=“seu-arquivo.js” async></script>
```

O atributo `async` avisa o navegador de que ele não precisa carregar o script no momento em que ele

é encontrado no código. Por isso, quando ele passa por lá, busca script enquanto carrega o resto da página. Esse atributo não é tão novo assim, mas ainda tem muita gente que não o usa por não saber que ele existe.

Há outro atributo chamado `defer`, que vai indicar para o browser que aquele determinado script só deve ser executado depois que o documento todo for carregado.

Não é muito complicado entender, mas existem 3 possibilidades de execução usando o `defer` e o `async`: quando o `async` está presente, o script será executado de forma assíncrona, assim que ele tiver sido carregado. Quando o `defer` for definido, mas o `async` não, o script só será executado quando a página tiver terminado de carregar. Se nenhum desses atributos estiver presente, o browser faz o download do script e o executa imediatamente, antes de terminar de carregar o resto da página.

Isso é explicado de forma detalhada na documentação do W3C: http://bit.ly/w3cdefer.

Segue uma imagem muito autoexplicativa. Ele mostra como funciona estas possibilidades:
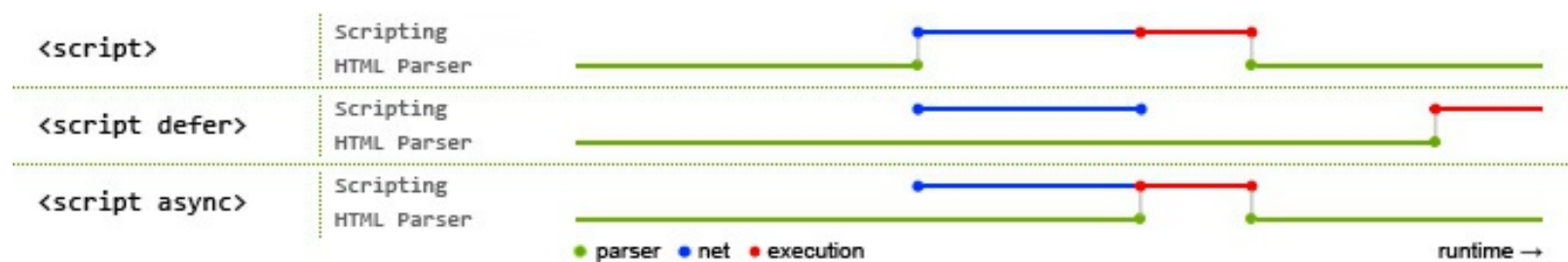
Fig. 8.1: Imagem tirada do artigo Asynchronous and deferred JavaScript execution explained de Peter Beverloo: http://bit.ly/1ywKn2v

8.4 Para estudar mais

O que você leu até agora é o caminho das pedras para entender melhor como os browsers funcionam. Na verdade, essa é apenas uma introdução, incompleta para ser sincero. Esse assunto é muito interessante e você acaba se apaixonando por ele com o tempo. Geralmente, a maioria dos desenvolvedores ignora toda essa teoria, mas você já não é mais maioria, é? ;-)

Procure sobre estes assuntos para estudar:

- GZip
- Compilação, Minificação e concatenação de CSS, JS e HTML http://csswizardry.com/2013/01/front-end-performance-for-web-designers-and-front-end-developers/#section:gzipping-and-minifying
- Prefetch: http://csswizardry.com/2013/01/front-end-performance-for-web-designers-and-front-end-developers/#section:resource-prefetching http://calendar.perfplanet.com/2012/speed-up-your-site-using-prefetching/
- Entenda como funciona a tag SCRIPT e seus atributos: http://www.w3.org/TR/html5/scripting-

1.html#the-script-element

- Otimização de imagens
- Imagens em WebP
- Elemento Picture
- Atributo srcset

Aqui vão mais alguns links interessantes:

- http://browserdiet.com/pt/
- http://tableless.com.br/performance-frontend-parte1/
- http://tableless.com.br/performance-frontend-parte2/
- https://developers.google.com/speed/articles/reflow
- https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction?hl=pt-br
- https://dev.opera.com/articles/efficient-JavaScript/?page=3#reflow
- http://www.phpied.com/css-and-the-critical-path/
- https://developers.google.com/web/fundamentals/

Acessibilidade

9.1 O que é acessibilidade para web?

Resposta rápida: é manter a informação acessível para qualquer pessoa, em qualquer condição, em qualquer dispositivo ou meio de acesso.

A ideia é que a informação fique ao alcance de qualquer um, não importa se essa pessoa tenha alguma deficiência ou não, se está usando ou não algum dispositivo, leitor de tela ou qualquer outra coisa que a ajude a consumir informação. Acessibilidade na web é tudo sobre manter a informação disponível para qualquer coisa ou qualquer um consumir na hora em que quiser.

Entenda que acessibilidade não é altruísmo. Você não mantém a informação acessível para leitores de tela como o JAWS ou NVDA só porque tem dó de pessoas cegas.

Até o momento da escrita desse livro, acessibilidade na web ainda é uma barreira em diversas empresas. Para a diretoria, faltam números para justificar. Para os desenvolvedores, falta tempo e interesse. Em empresas onde há um pouco de interesse, o assunto demora para se desenrolar até que haja alguma atitude de verdade.

O que poucos sabem é que a acessibilidade envolve UX, front-end, back-end, produção de conteúdo, QA e até marketing/vendas. Cada uma dessas áreas deveria ter um pouco de preocupação com este aspecto.

Acessibilidade na web é muito, mas muito interessante, mas é um assunto bastante extenso. Por isso, quero mostrar apenas alguns tópicos para que você estude mais daqui a pouco.

9.2 Acesso pelo teclado

O básico para qualquer sistema ou website é que sejam acessíveis via teclado. Uma suposição que fazemos todos os dias é de que qualquer visitante usa mouse. Essa é uma conclusão, no mínimo, equivocada. Sem entrar no quesito de que seus visitantes podem estar usando um smartphone ou tablet, é necessário que seu site seja facilmente navegável via teclado.

O teclado é ainda a segunda interface entre o seu produto e o usuário. Existem duas maneiras de preparar seu produto ou website para que sejam acessíveis via teclado: caminho de *tabs* e *AccessKey*. Vamos falar um pouco mais sobre a sequência de tabs, que eu julgo ser mais importante.

O caminho de tabs é simples: basta inserir o atributo `tabindex` nos elementos HTML como links, inputs e botões para que fiquem acessíveis ao apertar a tecla `TAB` do teclado. Veja o código a seguir:

```
1 <a href="#" tabindex="1">Primeiro link</a>
2 <a href="#" tabindex="3">Segundo link</a>
```

```
3 <a href="#" tabindex="2">Terceiro link</a>
```

Perceba os valores do atributo `tabindex`. Quando o usuário apertar `TAB`, o **primeiro link** ganhará foco. Ao apertar a tecla `TAB` outra vez, o **terceiro link** ganhará foco, já que o valor `tabindex` está como 2. Ao apertar `TAB` novamente, o `segundo link** ganhará foco, já que seu` tabindex%% é 3. Sabendo disso, você pode fazer uma sequência de tabs em seu sistema, levando o usuário para pontos estratégicos do seu produto. Isso é muito útil para telas com formulários, por exemplo, onde o usuário pode evitar a interação com mouse.

Há muitos usuários que têm alguma deficiência motora, com a qual é muito difícil manusear o mouse e a utilização do teclado se torna imprescindível. Os usuários com deficiência visual também utilizam bastante o teclado, já que o leitor de tela narra o conteúdo dos elementos que ganham foco, principalmente os links.

9.3 Input Types

O HTML5 iniciou seu projeto remodelando os campos de formulários do antigo HTML. Até a criação do HTML5, nós usávamos formulários da mesma forma desde o início da internet. Havia apenas um punhado de tipos de campos que eram possíveis de usar. Com o avanço da internet, esses campos se tornaram cada vez mais limitados e a necessidade de novos tipos de formulários foi inevitável.

Hoje, os novos campos não apenas ajudam no trabalho do desenvolvimento, mas se focam principalmente na facilidade para preenchermos diversas novas informações. Campos novos como: *date*, *number*, *email* ajudam muito em dispositivos com telas pequenas, modificando as teclas. Veja o exemplo a seguir, onde o teclado muda de acordo com o formulário. A imagem da esquerda não tem a marcação apropriada, por isso o teclado aparece com letras. Já na segunda imagem, o formulário foi marcado com o novo tipo de campo chamado *number*, fazendo com que mostrasse apenas números:

Fig. 9.1: Diferença de teclado do iPhone

Este é o link para a documentação oficial do W3C, onde há uma lista com todos os tipos de campos que já podem ser usados em nossos projetos: http://www.w3.org/TR/html5/forms.html#states-of-the-type-attribute.

Uma dica: quando o browser não suporta algum tipo desses novos campos, ele simplesmente mostra para o usuário o campo simples de texto (`type="text"`). Assim nada aparece quebrado e o usuário consegue seguir seu fluxo normalmente. Isso se chama *Fault Tolerance*. Mas aí já é assunto para outra hora.

9.4 WAI-ARIA

O HTML serve para apenas uma coisa: dar significado à informação. Ele faz isso marcando a informação com tags. Depois disso, você escreve seu CSS e o JavaScript, que ficarão com as responsabilidades de formatar essa informação e manipular seu comportamento.

O ponto é que a semântica não fica apenas na hora de marcar a informação com HTML puro. Como você faz para que um usuário com deficiência visual saiba que a informação que ele procura está dentro de uma tab, e essa tab está fechada? Como você avisa que aquele lugar onde ele está navegando é o lugar mais importante da página? Que aquele monte de links de que o leitor de tela está falando é o menu principal do site?

Estendendo o significado

A WAI-ARIA serve para estender o significado das interações do seu site. Quando as tags do HTML5 vieram, elas já começaram um trabalho importante de dar significado às estruturas do layout. Você consegue marcar o que é um menu de navegação, uma sidebar, um header, um footer etc. Esse trabalho é muito importante porque ajuda a definir a importância que cada elemento contém. Antes do HTML5 isso era impossível, tudo era DIV.

A semântica que as Microdatas trazem também são uma maneira de estender o significado do conteúdo. Ele pode ter informações importantes e que passam despercebidas pelos robôs de busca e outros meios de acesso.

A WAI-ARIA vai ajudar muito em aplicações onde a informação é dividida em várias porções na tela em diversos elementos que precisam de interações (como o clique) para que seja visualizada, fazendo com que a acessibilidade seja prejudicada e o usuário não consiga acessar todas as partes desse layout de maneira linear. A ideia é que toda e qualquer informação, escondida em uma modal, tab, collapses etc. seja acessível.

Roles, States e Properties

Para que você pesquise melhor quando for estudar o assunto, a WAI-ARIA divide a semântica em duas partes: *Roles*, que define que tipo de elemento o usuário está interagindo e *States/Properties* que são suportadas pelas Roles e definem o estado do elemento.

Com a Role você fala que determinado elemento é um *collapse*, e com os States/Properties você diz se essa collapse está aberta ou fechada. Isso tudo você vai definir direto no HTML, por meio dos atributos.

São 4 tipos de roles. Cada tipo de role é responsável por um determinado gênero de elemento.

- *Abstract* para definição de conceitos gerais. Não devemos usar para marcar conteúdo. Confesso que ainda estou tentando entender esse tipo de Role. “Conceitos abstratos” é algo muito abstrato para mim. :-)
- *Widgets* para marcar elementos de interface soltos, como caixas de alerta, botões, checkbox, links, tabs etc.

- *Document Structure* para definir estruturas de organização da página que não são interativas como o header, footer, sidebar, main etc.
- *Landmarks* para regiões de página que são pontos importantes para onde o usuário navegaria, por exemplo: buscas, conteúdo principal, sidebar, formulários etc.

Não dá para mostrar aqui todas as Roles que existem porque são muitas. Para ver toda a lista, vá direto ao site do W3C (http://www.w3.org/TR/wai-aria/roles#roles_categorization). Lá você vai encontrar todas as categorias.

Posso inserir WAI-ARIA via JavaScript?

Sim. Muitos até preferem colocar estes atributos via JavaScript. Não há problema algum. Insira os atributos em cada uma das suas funções de comportamento. Isso facilita o trabalho. Por outro lado, se você sabe que o seu público desativa o suporte de JavaScript do Browser (o que é muito, muito difícil), prefira colocar diretamente no código HTML do elemento. Assim você garante que o WAI-ARIA vai funcionar mesmo que o usuário desabilite o JS.

Eu tenho alguns slides de uma apresentação sobre o assunto, segue o endereço: https://www.slideshare.net/diegoeis/waiaria-interaes-acessveis-na-web.

Pré-processadores CSS

Imagine se você pudesse criar variáveis no CSS. Imagine também se você pudesse usar condições como `if` ou `while` no seu código CSS... Essas e outras funções são possíveis se utilizarmos um pré-processador.

Neste capítulo, veremos bastante sobre SASS, que é o pré-processador mais utilizado e mais completo que existe até agora. Eu também estou mais familiarizado com ele, já que o uso no meu dia a dia. Existem vários outros pré-processadores, como Stylus e Less, que não vou abordar aqui, mas na essência todos têm quase a mesma sintaxe.

10.1 Por que usar pré-processadores?

Um dos principais motivos seria automatizar e evitar repetição de código. Em alguns momentos, em qualquer linguagem, você precisa escrever blocos de código que podem se repetir em várias partes do projeto. E vamos ser sinceros: o CSS é uma linguagem um pouco burra quando se trata de lógica. Tudo é muito manual e não é possível automatizar a grande maioria das tarefas. A reutilização de código é mínima, o que nos obriga a lidar com código em excesso. É aí que a graça dos pré-processadores se destaca. A utilização de mixins, variáveis, funções e condições aproxima a lógica do código CSS.

Outro ponto é o tempo de manutenção. A partir do momento em que você começa a utilizar as features de um pré-processador, você precisa apenas se preocupar com pequenos blocos de código. As variáveis são um bom exemplo disso. Quem nunca teve problemas para mudar uma determinada cor hexadecimal e precisou mudar o valor arquivo por arquivo, linha por linha? Se você tivesse essa cor gravada em uma variável, bastava mudar o código hexadecimal em apenas um lugar e tudo estaria resolvido.

Mas a função principal de um pré-processador é que ele ajuda a gerenciar se CSS. Tudo nos pré-processadores é pensando em como você consegue melhorar mais seu código e como ele pode ficar mais organizado. Essa ideia começa na maneira como você organiza seus arquivos e vai até a forma com que você escreve seu código.

CSS é muito simples. Mas uma má organização pode levar equipes inteiras à loucura em longo prazo. Os pré-processadores ajudam a gerenciar as partes fracas do CSS, que ele não consegue resolver sozinho. É nisso que você tem que se focar. Não importa qual pré-processador você vai usar, mas tenha em mente que ele não é uma ferramenta para escrever menos CSS, mas para organizar seu CSS.

10.2 Um pouco de SASS

SASS é o pré-processador mais utilizado no momento. Existem outros como o LESS e o Stylus, mas o SASS que ganhou o gosto dos devs. No final de tudo, eles serão praticamente a mesma coisa. Algumas pequenas mudanças drásticas entre um e outro, mas a essência sempre será a mesma.

Hoje eu uso SASS em praticamente todos os meus projetos. Por isso, vale a pena tentar incluir no seu stack de desenvolvimento. Seguem alguns exemplos que podem ser encontrados em outros pré-processadores com algumas diferenças na sintaxe. Eles resolvem problemas reais, que temos todos os dias em projetos de todos os tamanhos.

Usando variáveis

Para entender melhor, pegue como exemplo o código a seguir:

```
.tema-azul body {
  background-color: #0176bb;
}

.tema-vermelho body {
  background-color: #e3413e;
}

.tema-amarelo body {
  background-color: #f8e042;
}
```

Repare que nós repetimos muito código nesse exemplo. Veja como faríamos isso com SASS:

```
$azul: #0176bb;
$vermelho: #e3413e;
$amarelo: #f8e042;

.tema-azul body {
  background-color: $azul;
}

.tema-vermelho body {
  background-color: $vermelho;
}

.tema-amarelo body {
  background-color: $amarelo;
}
```

Já melhoramos bastante separando os hexadecimais em variáveis, as quais podemos usar em qualquer lugar do documento. Mas nós podemos automatizar ainda a criação dos seletores. Veja o exemplo:

```
$cores: (
```

```
2   azul: #0176bb,
3   vermelho: #e3413e,
4   amarelo: #f8e042
5 );
6
7 @each $tema, $cor in $cores {
8   .tema-#{tema} body {
9      background-color: $cor;
10   }
11 }
```

O `$tema` seria cada uma das chaves, no nosso caso azul, vermelho e amarelo. O `$cor` seria cada um dos valores dos temas, ou seja, os valores hexadecimais. A função `@each` está dizendo assim: a cada valor dos temas (azul, vermelho, amarelo) que encontrar no mapa `$cores`, repita o bloco de código.

Veja que a quantidade de código escrita é equivalente e até maior do que a versão de CSS do primeiro exemplo. Mas pense na manutenção disso. Precisaríamos inserir uma nova linha e adicionar um hexadecimal novo para criar uma cor. A função `@each` vai tratar de criar os blocos de CSS com o nome do tema e o hexadecimal.

Escrevi um artigo que fala sobre como organizar temas de cores usando SASS em projetos complexos aqui neste link: http://tableless.com.br/utilizando-maps-sass/.

Condicionais

Mas o que me deixa mais empolgado com os pré-processadores é a possibilidade de usar condicionais. Funções como `if`, `else`, `while`, `each` e `for`, que normalmente vemos em outras linguagens, podem ser usadas agora no CSS por meio dos pré-processadores. Criar condições é incrivelmente útil em vários momentos do trabalho com CSS.

if

Se algo for qualquer coisa diferente de *falso*, execute determinado comando.

Um uso bastante simples é da própria documentação do SASS:

```
1 $type: monster;
2 p {
3   @if $type == ocean {
4     color: blue;
5   } @else if $type == matador {
6     color: red;
7   } @else if $type == monster {
```

```
8       color: green;
9     } @else {
10      color: black;
11    }
12  }
```

Como era de se esperar, existe um `@else` para nos dar todo o apoio. Esse exemplo foi bastante simples. Se você tem uma variável qualquer, no nosso caso `monster`, ela pode mudar de valor em determinado lugar do código. Se mudar, há uma série de condições ali.

Vamos para um exemplo mais real:

```
1 $color1: #f9f9f9;
2 p {
3   @if (lightness($color1) < 30) {
4     background-color: white
5   } @else {
6     background-color: black
7   }
8 }
```

Estamos medindo a quantidade de branco que há na variável `$color1`, que vai receber um código de cor. Se for menor que 30% de luz (claridade), background `black` no elemento, caso contrário, background `white`.

for

Inicia uma variável e executa uma ação, incrementando essa variável um determinado número de vezes. Para cada repetição, uma variável de contador é usada para ajustar o código de saída.

É muito útil para você fazer códigos como o seguinte:

```
1  .item-1 {
2    width: 10px;
3    font-size: 10px;
4  }
5  .item-2 {
6    width: 20px;
7    font-size: 20px;
8  }
9  .item-3 {
10   width: 30px;
11   font-size: 30px;
12 }
```

```
13 .item-4 {
14   width: 40px;
15   font-size: 40px;
16 }
```

Você faria uma condição assim:

```
1 @for $i from 1 through 4 {
2   .item-#{$i} {
3     width: 10px * $i;
4     font-size: 10px * $i;
5   }
6 }
```

Outro truque interessante é que, se você colocar o primeiro número maior que o segundo, assim: `@for $i from 4 through 1`, a função vai decrementar ao invés de incrementar a variável. O resultado:

```
 1 .item-4 {
 2   width: 40px;
 3   font-size: 40px;
 4 }
 5
 6 .item-3 {
 7   width: 30px;
 8   font-size: 30px;
 9 }
10
11 .item-2 {
12   width: 20px;
13   font-size: 20px;
14 }
15
16 .item-1 {
17   width: 10px;
18   font-size: 10px;
19 }
```

Simple like that.

each

Define uma variável para item de uma lista de valores, produzindo blocos de código utilizando os valores da lista.

Vamos a um exemplo rápido. Para um código assim:

```
1 .tema-azul body {
2   background-color: #0176bb;
3 }
4
5 .tema-vermelho body {
6   background-color: #e3413e;
7 }
8
9 .tema-amarelo body {
10   background-color: #f8e042;
11 }
```

Você teria um SASS assim:

```
1 $cores: (
2   azul: #0176bb,
3   vermelho: #e3413e,
4   amarelo: #f8e042
5 );
6
7 @each $tema, $cor in $cores {
8   .tema-#{tema} body {
9      background-color: $cor;
10   }
11 }
```

Você gerencia basicamente um bloco de CSS e as variáveis para adicionar ou remover cores.

while

Repete um determinado bloco e código enquanto determinado estado for verdadeiro.

O exemplo mais bacana que pode ser mostrado aqui é a criação de um grid básico. Para ter um código como o seguinte:

```
1 .column-grid-1{
2   width: 60px;
3 }
4
5 .column-grid-2 {
6   width: 150px;
7 }
```

```
 8
 9 .column-grid-3 {
10   width: 240px;
11 }
12
13 .column-grid-4 {
14   width: 330px;
15 }
16
17 .column-grid-5 {
18   width: 420px;
19 }
20
21 .column-grid-6 {
22   width: 510px;
23 }
24
25 .column-grid-7 {
26   width: 600px;
27 }
28
29 .column-grid-8 {
30   width: 690px;
31 }
32
33 .column-grid-9 {
34   width: 780px;
35 }
36
37 .column-grid-10 {
38   width: 870px;
39 }
40
41 .column-grid-11 {
42   width: 960px;
43 }
44
45 .column-grid-12 {
46   width: 1050px;
47 }
```

O código seria algo mais ou menos assim:

```
 1 $i: 1
 2 $column-width: 60px
 3
 4 @while $i < 13 {
 5   .grid-#{$i} {
 6     column-width: $column-width;
 7    }
 8   $column-width: $column-width + 90px;
 9   $i: $i + 1;
10 }
```

Claro, para fazer um grid um pouco mais complexo, usando porcentagens, guardando o valor do *gutter* em uma variável, *roas* etc. você precisa de um pouco mais.

10.3 Pontos para pensar

Eu sempre disse que era meio contra a utilização de pré-processadores. Mas confesso que depois de usá-los, vi muita vantagem no desenvolvimento diário. Há motivos para você pensar se é o caso de você adotar ou não algum pré-processador CSS no seu projeto.

O Miller Medeiros (http://blog.millermedeiros.com/about/) escreveu um ótimo artigo com o nome de *The problem with CSS pre-processors* (http://blog.millermedeiros.com/the-problem-with-css-pre-processors/). Lá ele cita alguns motivos interessantes sobre não usar pré-processadores. Sugiro que você leia.

Pré-processadores podem ajudar como também podem maltratar bastante o projeto. Basta um escorregão para que seu projeto vire um inferno. Mesmo assim eu incentivo que você experimente e tire suas próprias conclusões. Nas próximas linhas dou meus dois centavos sobre o assunto.

Curva de aprendizado de novos integrantes

É comum recebermos novos integrantes na equipe. Pessoas vêm e vão. Isso é normal em qualquer time. O problema é que sempre que alguém novo chega, ele precisa aprender de preferência o mais rápido possível os detalhes do projeto. Pra facilitar essa curva de aprendizado, as equipes podem manter um manual explicando as boas práticas de escrita de código. Mas não importa quantas reuniões você faça, o cabra só vai conseguir conhecer mesmo o tamanho do projeto quando colocar a mão na massa.

Quando o código foi escrito sem nenhum pré-processador, a curva de aprendizado é muito menor. Ele não precisa aprender os costumes da equipe quanto às preferências do uso das features do pré-processador escolhido. Se a equipe estiver usando SASS, por exemplo, pode existir uma série de regras de mixins e extends que podem demorar para fazer sentido. Sem comentar o aninhamento (*nesting*) de seletores para evitar o repetição de elementos. Já é difícil entender todas as relações de

elementos e suas formatações na maneira normal, imagine usando indentação para hierarquizar os seletores.

Isso pode prejudicar bastante a manutenção, principalmente se quem a fizer for um novato no projeto que, por não conhecer os miúdos do código, pode acabar alterando algum valor que é reutilizado em vários mixins/extends e seus correlatos.

Para evitar problemas assim, a documentação deveria ser muito bem atualizada, apontando todas as relações de mixins, extends, variáveis etc. E se você já se aventurou a tentar manter uma documentação atualizada, sabe que é quase impossível ter 100% de comprometimento para atualizar os mínimos detalhes. Principalmente detalhes que mudam o tempo inteiro.

O caminho de ida é fácil, o de volta nem tanto

Quando decidimos usar um pré-processador é um caminho sem volta.

Até onde pesquisei, uma vez que seu projeto foi escrito sob as regras de um pré-processador, não há como voltar atrás, a não ser que você utilize a versão já processada desse código. Mas aí entramos em outro problema: geralmente pré-processadores escrevem mais código do que você escreveria manualmente. Duas opções: corrigir o excesso de código do projeto inteiro na mão ou ignorar e seguir em frente.

Nunca será como se fosse escrito manualmente

Como já citamos, o código gerado pelo pré-processador nunca será como o código escrito por você manualmente. Nunca vi nada que tente gerar código automaticamente fazer algo que não fosse duvidoso.

Há equipes que se arrependem e decidem que deste ponto em diante escreverão CSS puro, sem utilizar as features do pré-processador. Mas aí perde toda a graça e não há vantagem nenhuma.

Veja um exemplo simples de uso do *mixin* do SASS:

```
1  @mixin alert {
2      color: red;
3      border: 2px solid red;
4  }
5
6  .alert-default {
7      @include alert;
8  }
9
10 .alert-special {
11     @include alert;
12     background-color: gray;
```

```
13 }
```

Resultado:

```
1 .alert-default {
2   color: red;
3   border: 2px solid red;
4 }
5
6 .alert-special {
7   color: red;
8   border: 2px solid red;
9   background-color: gray;
10 }
```

Entende a duplicação código de `border` e `color` das duas propriedades? Claro, no exemplo acima seria muito melhor ter usado um `extend` em vez de `mixin`. Mas e se um desavisado cometer esse erro?

Usando `extend`, fica assim:

```
1 .alert {
2     color: red;
3     border: 2px solid red;
4 }
5
6 .alert-default {
7     @extend alert;
8 }
9
10 .alert-special {
11     @extend alert;
12     background-color: gray;
13 }
```

Que já pode ser compilado para:

```
1 .alert, .alert-default, .alert-special {
2     color: red;
3     border: 2px solid red;
4 }
5
6 .alert-special {
7     background-color: gray;
```

```
8 }
```

Cedo ou tarde o CSS vai amadurecer

O W3C tem trabalhado bastante para oficializar o mais rápido possível as novidades do CSS. Os fabricantes de browsers também têm se esforçado para manter seus navegadores atualizados, trazendo sempre novidades para experimentarmos e usarmos em projetos hoje, agora. Cedo ou tarde o CSS vai ter features como as dos pré-processadores para nos ajudar com a manutenção e economia de código. Não é algo imediato, claro, mas sabemos que virá.

Com certeza vai ser bem melhor do que temos hoje em vários sentidos. Por isso você pode esperar pra ver.

Concluindo

Eu uso os pré-processadores sempre que posso. Em projetos pequenos ou grandes, não importa. Hoje meu processo envolve usar *Middleman* e *SASS* nos projetos do Tableless. Isso me ajudou muito nos projetos, principalmente nos mais complexos. Para projetos com poucas pessoas ou nos quais apenas eu estou envolvido, ter o controle de tudo é muito importante.

Tente experimentar. Fiz um artigo no Tableless que explica como instalar SASS na sua máquina de maneira fácil. Segue o link: http://tableless.com.br/instalando-sass-na-maquina-video/

Com certeza, depois que você ultrapassar a curva de aprendizado, você nunca mais vai querer escrever CSS puro.

Ferramentário (Tooling)

Durante muito, mas muito tempo, nós não tínhamos ferramentas que nos auxiliassem no desenvolvimento front-end. Lembro-me que muitos desenvolvedores vibraram quando a extensão Web Developer Toolbar foi lançada no Mozilla/Firefox, pois ela possibilitava executar uma série de pequenas tarefas cotidianas. Eu não sei qual dos browsers lançou o Web Inspector, muito provavelmente o Opera, mas durante muito tempo o Inspector foi nossa única ferramenta além do editor de texto e mesmo assim era bastante limitada.

Hoje a história é outra. Existem centenas de ferramentas que medem a velocidade do site, que testam seu código, testam o layout em diversos browsers e sistemas operacionais, a responsividade do layout etc. Elas facilitam o seu trabalho e economizam um tempo precioso para descobrir e resolver problemas, mas principalmente para melhorar o workflow do trabalho.

A minha opinião é de que você não precisa aprender todas as novas ferramentas e tecnologias que existem por aí. Você pode muito bem se virar com algumas e aprender coisas novas de acordo com a sua necessidade. As ferramentas que citarei aqui são as mais populares. Algumas delas precisam de *ruby* ou *node.js* instalados na sua máquina.

Alerta

Não se sinta obrigado a aprender todas as ferramentas da moda. Muitos devs perdem tempo precioso dando atenção para novas ferramentas que nem serão usadas pela equipe ou que morrerão nos próximos meses. Não caia na ladainha de aprender tudo que aparece por aí.

Segue a dica de algumas ferramentas que estão algum tempo no mercado e que são úteis para muitos desenvolvedores. Mas, como disse, veja sua real necessidade de aprendê-las e não fique triste se você sentir que alguma delas não lhe será útil. Eu mesmo não uso Bower ou Yeoman.

11.1 Grunt

Depois que você termina de escrever código, há uma série de tarefas que precisam ser executadas para que esse código seja publicado em produção. São tarefas repetitivas que, se forem feitas manualmente, podem tomar muito do seu tempo. É aí que o Grunt aparece.

O Grunt é um automatizador de tarefas

Ele roda via linha de comando no terminal e tem o Node.js como base.

Automatizadores de tarefas não são novos. Só para você saber, várias outras linguagens já tinham suas soluções de automatizadores de tarefas. Para citar alguns: Make no Shell, o Ant no Java e o

Rake no Ruby.

Se você é bastante novo na área, talvez você não esteja acostumado com esse workflow de trabalho, mas geralmente desenvolvemos os projetos em um ambiente montado na nossa própria máquina antes de colocar esse código em produção para os usuários. Por isso, imagine o processo: em produção, o código é concatenado (todos os arquivos de JavaScript e de CSS são unificados em um só arquivo), ofuscado (resumindo: os nome de variáveis são trocados por nomes menores, geralmente para apenas uma letra) e minificado (código sem comentários, sem espaços, sem quebras de linha). Esses três processos não podem ser feitos manualmente. O Grunt entra como aliado, fazendo essas e outras tarefas.

Instalando o Grunt

O Grunt é instalado via NPM, que é o gerenciador de pacotes do Node.js. Se você ainda não tem Node.js, descubra como instalar no próprio site do Node. É simples, basta baixar e instalar na máquina (http://nodejs.org/download/).

Feito isso, você vai instalar o comando do Grunt em sua máquina com apenas um comando no terminal:

```
1 npm install -g grunt-cli
```

Essa instalação permite que você rode o comando `grunt` de qualquer diretório da máquina.

Rodando Grunt em um projeto que já existe

Se você já tiver em um projeto que tem Grunt instalado, pode seguir estes três passos simples que a documentação explica:

1. Vá para o diretório do projeto que usa Grunt;
2. Instale as dependências usando os plugins com `npm install`;
3. Rode o comando `grunt`.

Isso vai executar todas as tasks configuradas para disparar por default quando rodamos o comando `grunt`.

Agora, se você quer fazer algo do zero, vamos para o próximo passo.

Rodando Grunt do zero

Primeiro, você precisa fazer um setup, onde você vai declarar quais são as dependências e as informações de que seu projeto vai precisar para rodar as tarefas do Grunt. Esse arquivo é chamado de `package.json`.

O `package.json` é um arquivo usado pelo NPM para guardar as informações do projeto. Você irá

listar todos os plugins dentro desse arquivo, na opção `devDependencies`.

```
{
  "name": "LocawebStyle",
  "description": "A front-end framework of behavior and style.",
  "homepage": "http://locaweb.github.io/locawebstyle",
  "keywords": [],
  "author": "Locaweb Front-end Team <frontend@locaweb.com.br>",
  "contributors": [],
  "devDependencies": {
    "grunt": "^0.4.5",
    "grunt-contrib-jasmine": "^0.8.2",
    "grunt-contrib-jshint": "^0.11.0",
    "grunt-contrib-watch": "^0.6.1",
    "grunt-jasmine-runner": "^0.6.0",
    "jasmine-jquery": "^2.0.6"
  },
  "repository": {
    "type": "git",
    "url": "git://github.com/locaweb/locawebstyle.git"
  },
  "version": "3.7.0"
}
```

Você vai colocar esse arquivo dentro do Root do seu projeto. Juntamente com outro arquivo chamado `Gruntfile.js`. Rodando o `npm install` no mesmo diretório que esse arquivo, ele vai instalar todas as dependências necessárias.

Existem poucos passos para você criar um `package.json`:

- Rodando o `grunt-init`, será criado automaticamente um `package.json` específico para seu projeto;
- Rodando um `npm init`, será criado um arquivo básico;
- Ou pode-se iniciar com o código de exemplo a seguir, como a própria documentação recomenda:

```
{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-nodeunit": "~0.4.1",
```

```
 8     "grunt-contrib-uglify": "~0.5.0"
 9   }
10 }
```

Quando você quiser instalar um novo plugin, basta usar o comando `npm install <nome do plugin> --save-dev`.

A flag `–save-dev` instala o plugin e já vai gravar a dependência corretamente no `devDependencies`.

Feito isso, é hora de ir para o `Gruntfile.js`.

Criando o Gruntfile.js

O arquivo `Gruntfile.js` (que pode ser chamado de `Gruntfile.coffee`, se você usar CoffeScript), é composto por algumas partes:

- Uma função que envolve tudo, chamada de função de "wrapper";
- Configuração do projeto e das tarefas;
- Carregamento dos plugins e das tarefas;
- Tarefas customizadas.

Um exemplo da documentação mostra como seria um `Gruntfile.js` configurado para rodar o Uglify, uma task que minifica e concatena seu JavaScript.

```
 1 module.exports = function(grunt) {
 2
 3   // Project configuration.
 4   grunt.initConfig({
 5     pkg: grunt.file.readJSON('package.json'),
 6     uglify: {
 7       options: {
 8         banner: '/*! <%= pkg.name %>
 9         <%= grunt.template.today("yyyy-mm-dd") %> */\n'
10       },
11       build: {
12         src: 'src/<%= pkg.name %>.js',
13         dest: 'build/<%= pkg.name %>.min.js'
14       }
15     }
16   });
17
18   // Load the plugin that provides the "uglify" task.
19   grunt.loadNpmTasks('grunt-contrib-uglify');
```

```
20
21   // Default task(s).
22   grunt.registerTask('default', ['uglify']);
23
24 };
```

A função wrapper é essa aqui:

```
1 module.exports = function(grunt) {
2   // Aqui vai o resto do código
3 };
```

Para habilitar as tasks, você precisa carregar o plugin que provê a determinada task. Suponha que você queira carregar essa tarefa de Uglify, a linha a seguir é inserida no final do projeto, como no nosso exemplo inicial:

```
1 grunt.loadNpmTasks('grunt-contrib-uglify');
```

Feito isso, você já consegue rodar no terminal um `grunt uglify` no seu projeto. Ele rodará toda a configuração determinada dentro da tarefa `uglify`.

Conforme você vai inserindo muitas tarefas, é comum querer fazer tasks customizadas. Por exemplo, fazer uma tarefa que rode apenas os testes de JavaScript com o comando `grunt test` seria algo assim:

```
1 grunt.registerTask('test', ['uglify, jasmine']);
```

Não vou me aprofundar muito aqui porque seria reescrever uma tradução da documentação, por isso aconselho que você leia a documentação. É bastante simples.

O Grunt é uma ferramenta que vale a pena usar. Eu recomendo dar uma olhada cuidadosa na apresentação do Almir Filho (http://bit.ly/gruntocara). Você vai entender toda a facilidade que o Grunt pode trazer para o seu workflow. Veja alguns links interessantes para você estudar:

- Documentação oficial: (http://gruntjs.com/getting-started)
- Veja os slides da apresentação que o Almir Filho fez no link (http://bit.ly/gruntocara)
- Artigo do Tableless do Vagner Santana: *GRUNT — Você deveria estar usando* (http://tableless.com.br/grunt-voce-deveria-estar-usando/)
- Automação de Build de Front-end com Grunt.js (http://bit.ly/automacaogrunt)
- Site oficial com a documentação (http://gruntjs.com)

11.2 Bower

Não é incomum usar scripts e frameworks de terceiros em nossos projetos. Dependendo do trabalho,

se você não for organizado, esse código fica bem difícil de gerenciar. Alguns deles precisam ser atualizados ou você precisa baixar outras dependências. O Bower foi criado pelo pessoal do Twitter para ajudar o desenvolvedor a gerenciar todas as suas dependências.

Você sabe que qualquer desenvolvedor tem preguiça de atualizar bibliotecas, por menores que elas sejam. Quem nunca deixou o jQuery desatualizado por anos na pasta do projeto por pura preguiça de baixar a nova versão, entrar no projeto, apagar a versão velha e copiar a nova? Por que não automatizar esse processo?

O Bower é um gerenciador de pacotes client-side. É baseado em node.js e como o Grunt, roda via linha de comando. Se você já tem o NPM na máquina, basta ir no terminal e digitar `npm install -g bower`. Depois é correr para a alegria instalando suas dependências. Coisa simples assim: `bower install jquery` e pronto.

Há também um arquivo chamado `bower.json` que é o seu manifesto de dependências, ou seja, quando o Bower for executado, ele vai instalar automaticamente tudo o que está listado no `bower.json`. Segue um exemplo:

```
1 {
2   "name": "app-name",
3   "version": "0.0.1",
4   "dependencies": {
5     "sass-bootstrap": "~3.0.0",
6     "modernizr": "~2.6.2",
7     "jquery": "~1.10.2"
8   }
9 }
```

Uma explicação rápida de cada ponto ali:

- *name* é o nome da sua aplicação.
- *version* é a versão do sua aplicação.
- *dependencies* são os pacotes de dependências necessárias para sua aplicação rodar.

A coisa toda é bastante simples, mas vale a pena ler mais para se aprofundar:

- Slides de uma apresentação do Nando Vieira: (http://bit.ly/usandobower)
- Bower na Prática (http://tableless.com.br/bower-na-pratica/)
- Configurando o Bower e o Polymer (http://tableless.com.br/configurando-bower-e-polymer-2/)
- Site oficial do Bower no link (http://bower.io)

11.3 Dev Inspector

Todos os browsers têm uma ferramenta útil e indispensável chamada Inspector. O Inspector é a ferramenta mais usada para debugar o seu código final, gerado pelo browser. Hoje, a maioria dos browsers tem sua própria ferramenta de inspeção. O Firefox incorporou a Firebug (https://addons.mozilla.org/pt-br/firefox/addon/firebug/), o IE tem a Developer Tool (http://msdn.microsoft.com/en-us/library/dd565628(v=vs.85).aspx), o Chrome e o Safari têm o web inspector (http://trac.webkit.org/wiki/WebInspector) do próprio Webkit. O Opera usa o Dragonfly (http://www.opera.com/dragonfly/).

O Inspector é útil e talvez o mais útil de todas as ferramentas que você pode encontrar. Com ele você analisa, debuga, modifica, testa, e monitora seu código client-side. Não há como desenvolver websites hoje em dia sem a ajuda do Inspector. Ele vai lhe mostrar como o código é exatamente renderizado nos browsers e, se precisar fazer alguma modificação, ele mostrará o resultado na hora, direto no browser.

Você vai conseguir modificar o CSS, mexer no HTML, debugar o JavaScript e ter resultados instantâneos, verificando todos os efeitos das modificações no layout. Cada browser tem um Inspector com uma característica diferente. O Inspector do Chrome, por exemplo, é ótimo para analisar performance.
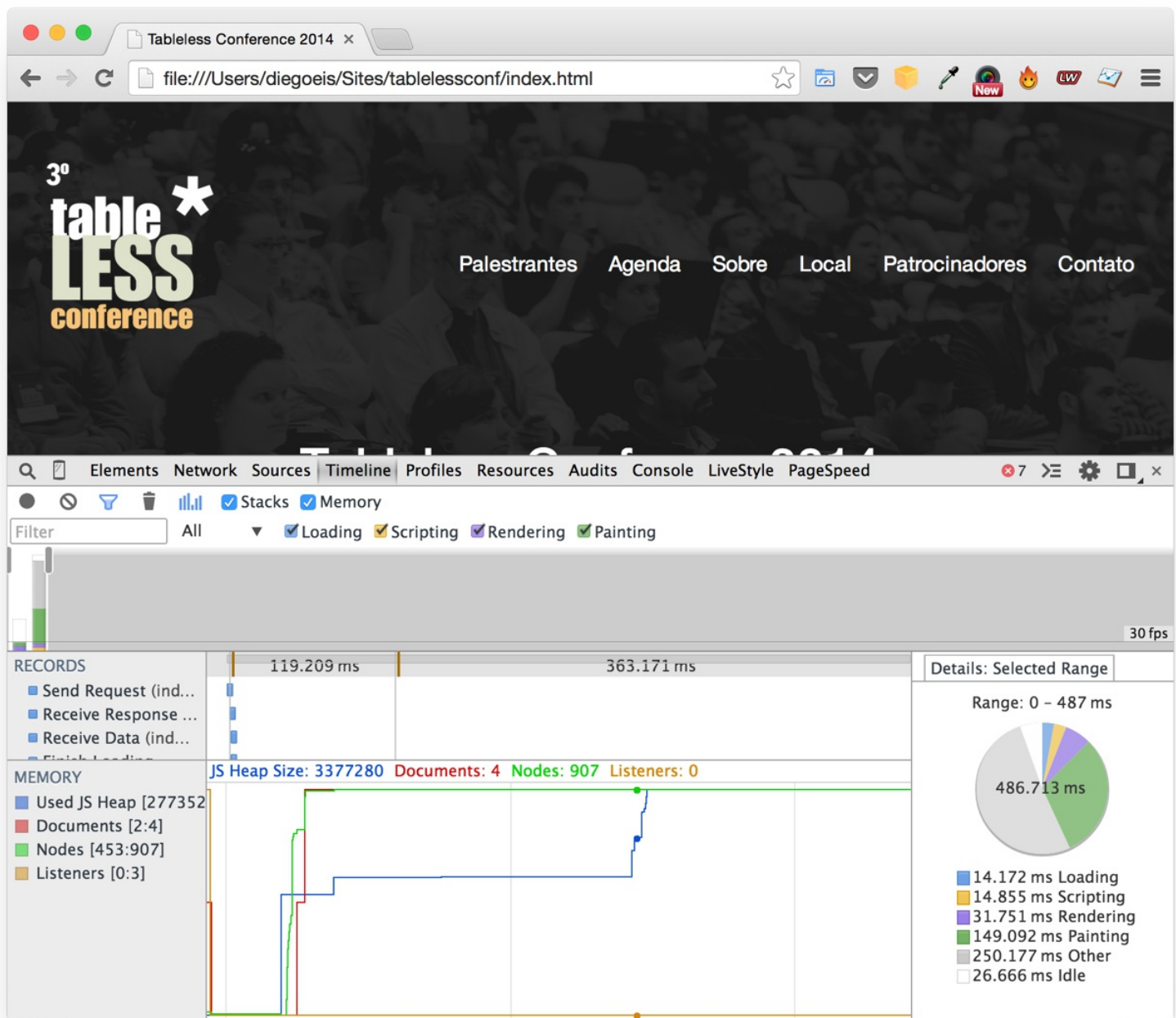
Fig. 11.1: Timeline do Chrome Dev Tools

Já no do Firefox, há uma análise do site onde ele representa os elementos do layout em formato 3D. Muito útil em websites complexos.

Fig. 11.2: Representação 3D do Firefox para facilitar o debug do layout.

O Inspector é o responsável por fazer você usar um browser para desenvolver e outro browser para navegar na internet como um usuário normal. Diversos desenvolvedores usam o Chrome como browser de desenvolvimento e o Firefox como browser padrão para navegar.

Leia mais sobre o Inspector:

- Page Inspector no MDN: https://developer.mozilla.org/pt-BR/docs/Tools/Page_Inspector
- Ferramentas e sites para tornar mais ágil o seu desenvolvimento: http://bit.ly/1xxphTM
- Meet your Web Inspector: http://ruby.bastardsbook.com/chapters/web-inspecting-html/

11.4 Console dos browsers

Todos os Inspectors dos browsers têm uma ferramenta chamada Console. O Console serve para debugar de forma mais específica o seu código JavaScript e também o DOM do HTML. Existem vários comandos e truques usando o console, não vou falar sobre todos. No final desta seção há uma lista de links para estudos, indicando outros truques e comandos.

console.log()

Vou me basear no Chrome, que é um dos browsers mais usados pelos devs. Para abrir o console no Chrome use o atalho (no mac) `cmd + option + J`. Ou clique na tela do browser com o botão direito, vá até *Inspect Element* e clique na aba *Console*.

O `console.log()` vai retornar uma mensagem no console. Simples e fácil. Vamos fazer um arquivo HTML simples como o seguinte. Depois de criado, abra-o no seu browser e abra o console para visualizar os resultados.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Teste de Console</title>
5  </head>
6  <body>
7
8  <p>Testando o console. Show!</p>
9
10 <script>
11 (function(){
12
13   console.log('Hello Console!');
14
15 })();
```

```
16 </script>
17
18 </body>
19 </html>
```

Ele apenas mostrou a mensagem que colocamos como parâmetro:
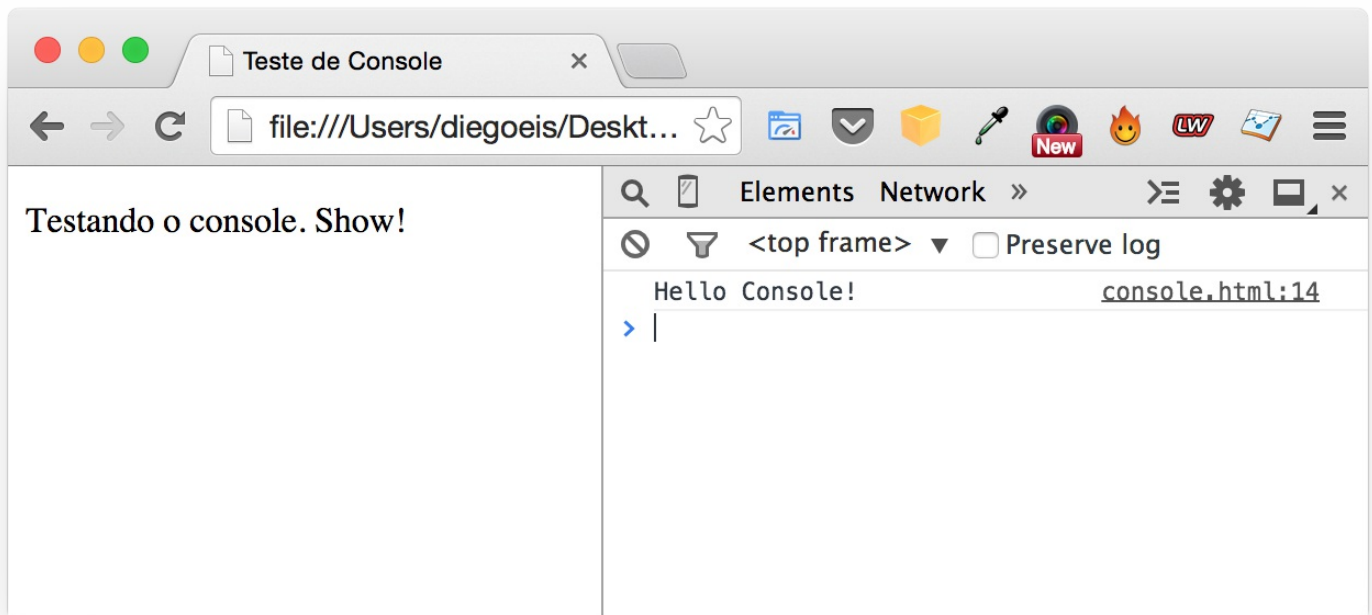
Fig. 11.3: Mensagem básica no console.

Você pode mostrar o valor de uma variável se quiser:
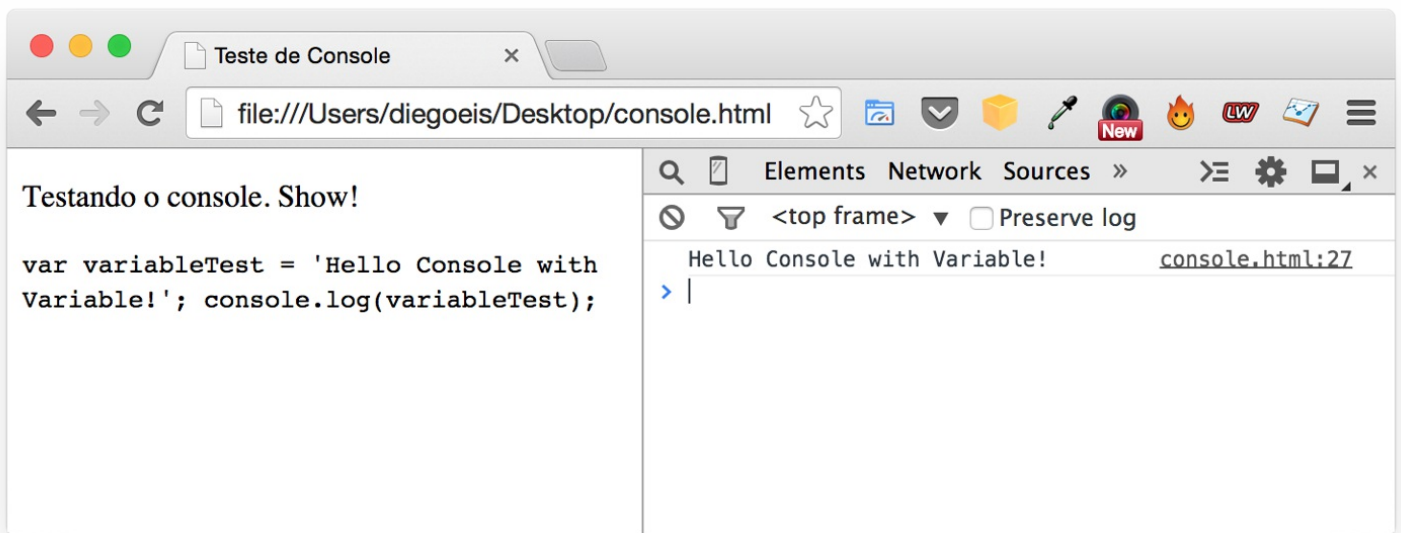
Fig. 11.4: Exibindo uma variável no console.log()

Você pode também concatenar uma string usando vários tipos de valores em apenas uma chamada de `console.log`:

```
1 var userName = 'Diego Eis';
2 var userAge = 30;
3
4 console.log('O usuário %s tem %d anos.', userName, userAge);
```

O resultado é este aqui:
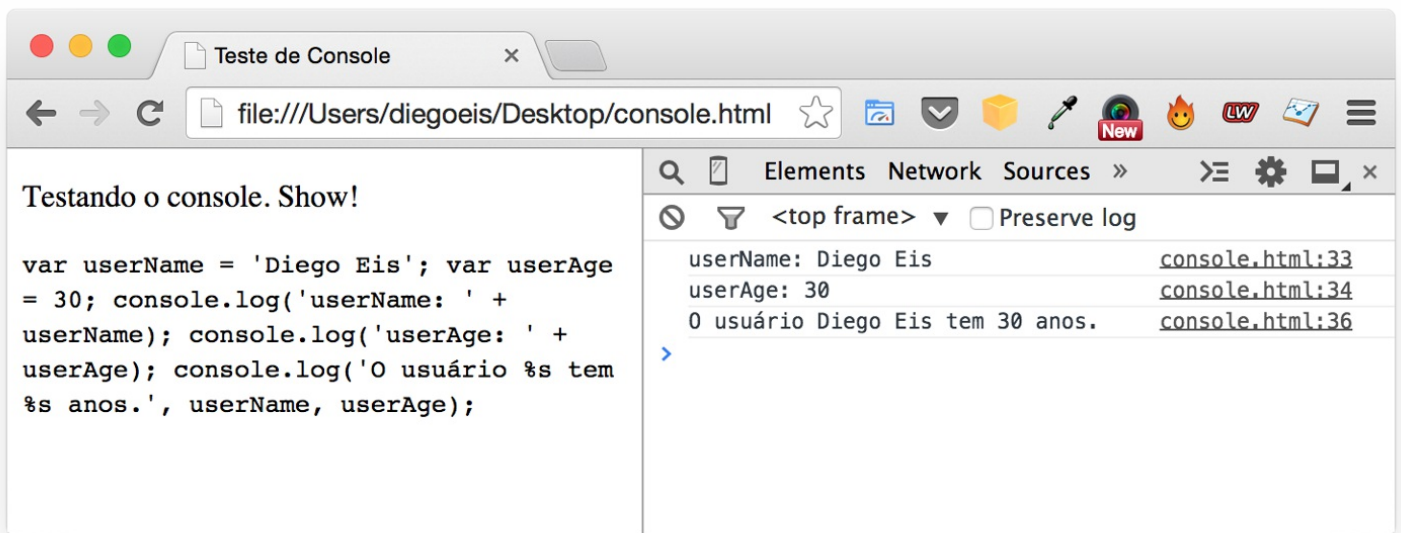
Fig. 11.5: Console.log exibindo resultado de concatenação de variáveis.

Note que o parâmetro `s` formatará seu valor como uma string. O parâmetro `d` formatará o valor como um inteiro. Claro, nem preciso dizer que só vai servir para números.

console.info() e console.error()

Estes dois funcionam da mesma maneira que o `console.log()`, mas eles têm uma função mais descritiva. Geralmente, usamos o `console.log()` para mostrar qualquer tipo de mensagem e muitas vezes misturamos mensagens de erro e as informativas no console. O `console.error()` e o `console.info()` servem para diferenciarmos as mensagens de erros e informativas das mensagens normais. Veja adiante:

```
1  function isEqualFour(){
2    var number1 = 2;
3    var number2 = 1;
4
5    console.log('A soma deve ser igual a 4.', 'Variável 1:',
6                number1, 'Variável 2:',number2);
7
8    if(number1 + number2 == 4) {
9      console.info('CORRETO! O valor é 4!');
10   } else {
11     console.error('ERRADO! A soma dos valores é ' +
12                   (number1 + number2) + '.');
13   }
```

```
14
15 }
16 isEqualFour();
```

Veja como ficaria a saída no console usando o `info()`:
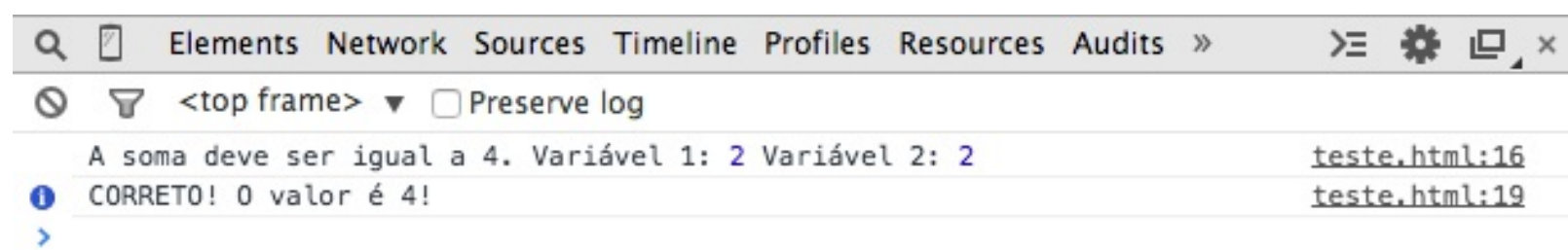
Fig. 11.6: O console trata o console.info() como uma mensagem de informação.

Agora o `console.error()`:

Fig. 11.7: O console trata o console.info() como uma mensagem de erro.

No Locaweb Style, o framework front-end da Locaweb (http://locaweb.github.io/locawebstyle/), nós usamos o `console.info()` para informar ao dev curioso quais módulos foram inicializados no carregamento da página:
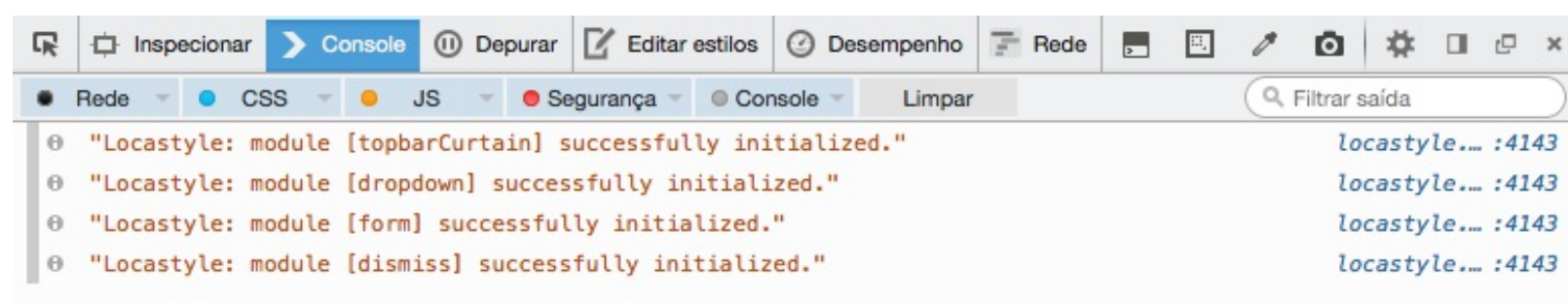
Fig. 11.8: Exemplo de console.info() usado no Locaweb Style

console.count()

O `count()` do console é muito útil. Em algumas ocasiões, você precisa saber quantas vezes uma função foi executada ou quantas vezes o script passou por algum lugar específico da sua função. Em vez de inserir uma série de `console.log()` no código e ficar contando depois no console, basta usar o `console.count()`.

O código (horrível) a seguir teria o retorno visto como na imagem:

```
 1 function teste(){
 2   console.count('Chamando a função teste()');
 3 }
 4
 5 function chamaTeste() {
 6   teste();
 7 }
 8
 9 function init(){
10   teste();
11   chamaTeste();
12 }
13 init();
```
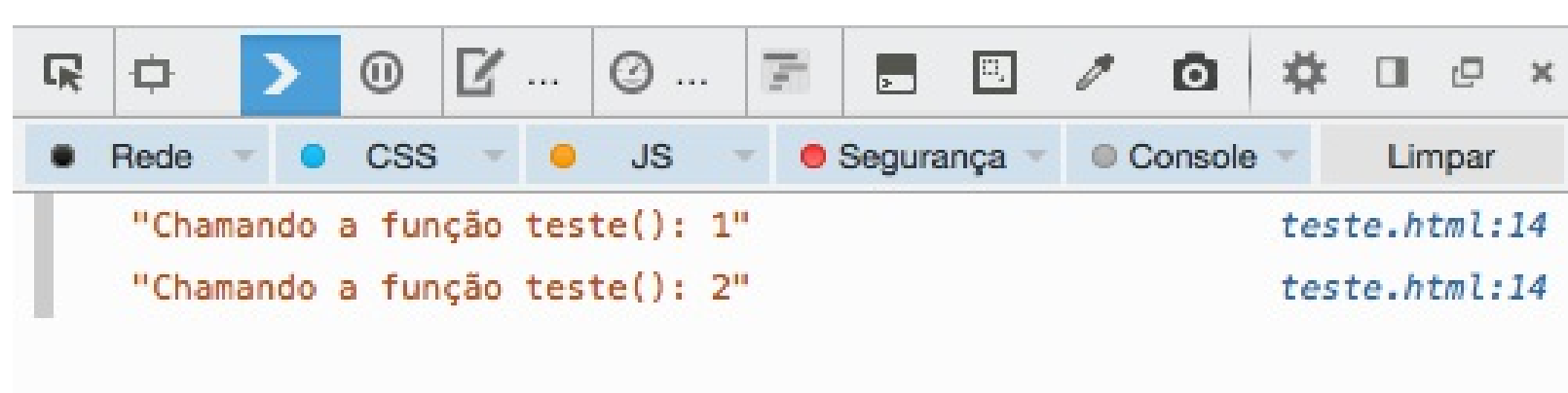
Fig. 11.9: Como o console.count() é exibido.

console.dir()

O `console.dir()` vai retornar uma representação de um objeto DOM como se fosse um objeto JavaScript. Basta digitar no console ou usar dentro de sua função da mesma forma que o `console.log()`, só que, como parâmetro, você passará um objeto do DOM. Veja um exemplo, onde mostrei o retorno da representação do `BODY` digitando o comando `console.dir(document.body)`:
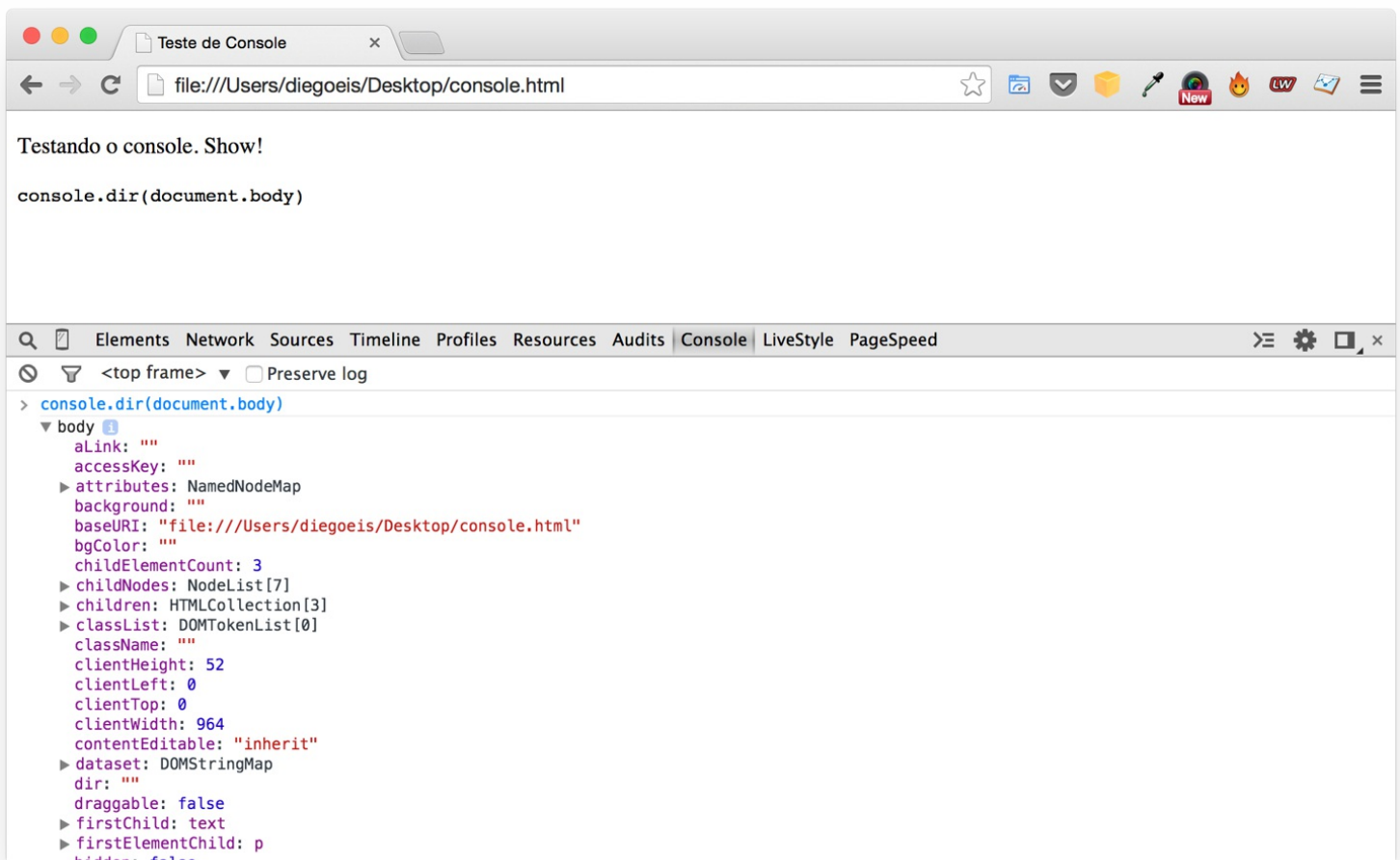
Fig. 11.10: Output de como o console trata a função console.dir()

Monitorando eventos via console

Há também uma função do console muito interessante chamada `monitorEvents`. Essa função ajuda a monitorar eventos, dando informações específicas sobre os elementos e sobre os eventos quando eles são disparados. Abra seu browser console e experimente o comando `monitorEvents(window, "resize”);`. Você vai monitorar o evento `resize` do elemento `window`. Toda vez que a janela é redimensionada, o evento é disparado e as informações sobre ele são mostradas no console. Veja a imagem a seguir como seria o resultado no Chrome:
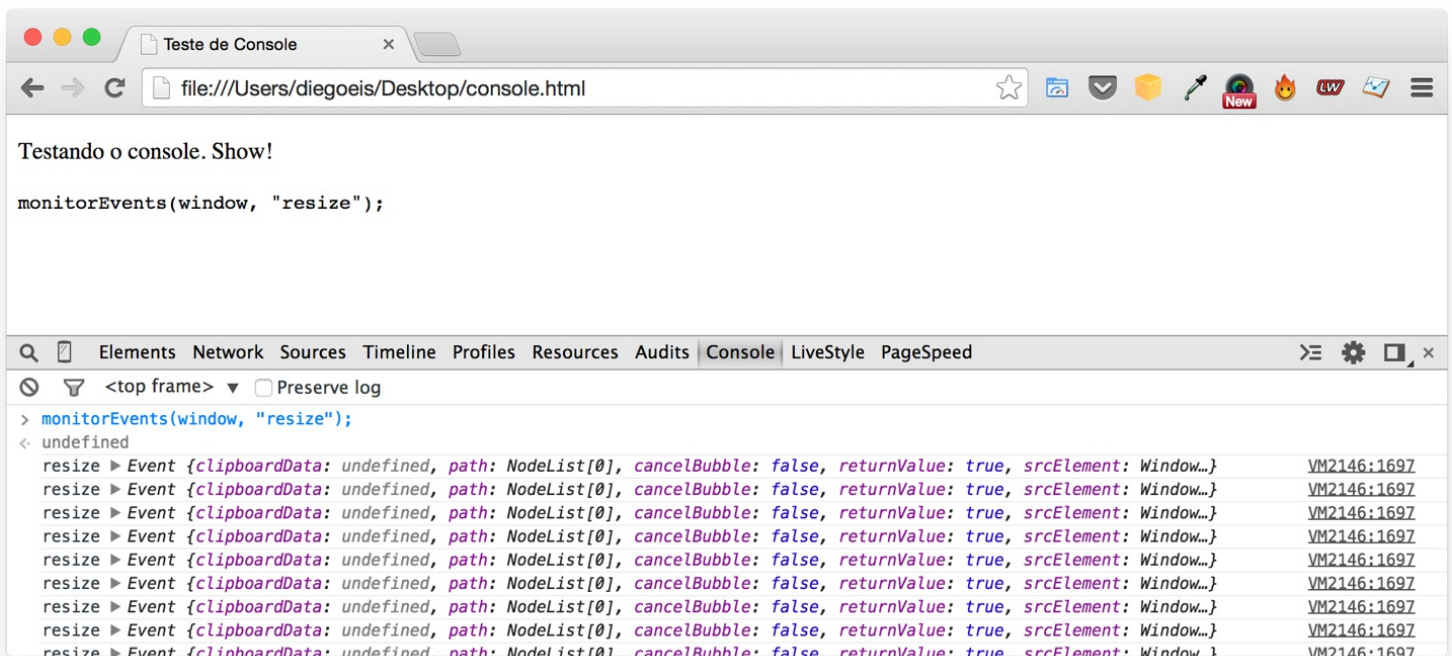
Fig. 11.11: Console listando o evento resize todas as vezes em que ele acontece.

Você também pode passar um array de eventos para um mesmo elemento. Por exemplo, monitorar os eventos `mouseup` e o `mousedown` no `body`:
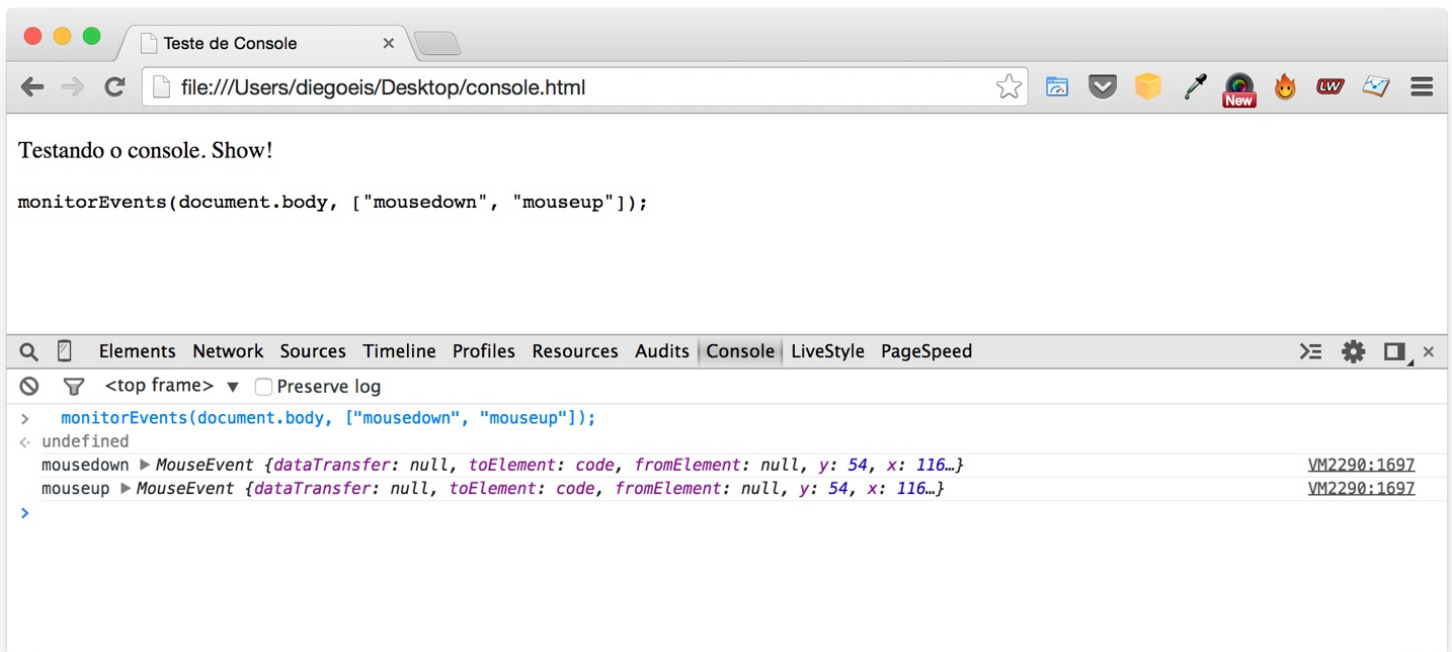
Fig. 11.12: Agora mostrando todas as vezes em que os eventos mouseup e mousedown acontecem no Body.

São comandos muito simples e que nos trazem uma grande ajuda durante o trabalho do dia a dia.

Existem vários outras funções do console tão úteis como essas. Veja a lista completa aqui: https://developer.chrome.com/devtools/docs/console-api

Muito simples e prático. Segue uma lista com mais links para você se esbaldar:

- Chrome Tips and Tricks — Console: https://developer.chrome.com/devtools/docs/tips-and-tricks
- Using the Console: https://developer.chrome.com/devtools/docs/console
- Firebug Console Tips: http://www.sitepoint.com/firebug-console-tips/
- How to debug your jQuery code: http://msdn.microsoft.com/en-us/magazine/ee819093.aspx
- Breakpoint actions in JavaScript: http://www.randomthink.net/blog/2012/11/breakpoint-actions-in-JavaScript/
- Commando Line Inspect Object: https://developer.chrome.com/devtools/docs/commandline-api#inspectobject

11.5 Sobre o uso do terminal

Escrever em algum terminal é sempre um desespero para os iniciantes. Mas saiba que sem ele você não vai muito longe. Você até consegue se virar durante um tempo, mas se você realmente quer experimentar novas tecnologias na área de front-end, vai precisar se aventurar no terminal.

Se você quiser usar algum pré-processador, rodar testes, ter prática com GIT, instalar geradores de arquivos estáticos como o Middleman ou usar coisas como Yeoman, Grunt, Gulp, Jasmine etc., você vai precisar meter a mão no terminal. E sabe de uma coisa? Depois que você se acostuma, você se sente na Matrix.

Outro motivo é que dependendo do projeto que você se envolver, principalmente se for em Ruby, Python e coisas desse tipo, o terminal será indispensável. Desde iniciar o projeto, debugar alguns erros, atualizar dependências e várias outras tarefas serão recorrentes. Na maioria das vezes, você vai precisar acessar o terminal para configurar e manter seu ambiente funcionando.

Ele não é difícil de usar. No começo, você pode até se assustar quando vê alguém mais experiente fuçando aquele monte de letras na tela, mas não se amedronte, na maioria das vezes o cara só está escaneando a tela procurando por palavras-chaves ou alguma mensagem de erro.

Mas eu tenho uma coisa triste para dizer: existem poucos tutoriais em português que ensine o básico para iniciantes. Neste link (http://bit.ly/1u3EyKN) você encontra os comandos básicos.

Eu realmente sugiro que você tente perder o medo de abrir o terminal e tente estudá-lo. Por enquanto, fica registrada aqui a importância do terminal para sua carreira. Este link vai ajudar muito no início:

https://linux.ime.usp.br/~lucasmmg/livecd/documentacao/documentos/terminal/Terminal_basico.html.

Diferença entre Shell e Bash

Você já pode ter se confundido, mas lá vai: Shell é um interpretador de comandos entre o computador e o usuário. Você tem vários tipos de interfaces para interagir com o Shell, como o Bash, escrito pelo Brian Fox para o projeto GNU.

Em uma analogia bem básica: é como se o Shell fosse o sistema operacional e o Bash fosse a sua interface gráfica.

Você pode usar vários tipos de interpretadores Shell, sendo que o `tcsh` é o que vem por padrão nos Macs. Para você, agora, nesse início, não se apegue a isso. Os comandos básicos serão os mesmos entre as interfaces e por enquanto você não vai sentir diferença.

- Apresentação de introdução aos comandos do Linux — http://bit.ly/1BPstNw
- Diferença entre Terminal, Shell, TTY e Console — http://bit.ly/1wSSvye

Produzindo sites com código estático

O primeiro problema que temos quando iniciamos a produção de um site é como resolver as partes de layout que são repetidas em todas as páginas do projeto, por exemplo: header, footer, sidebar e essas coisas. Já vi vários dos meus alunos mantendo sites com dezenas de arquivos HTML, sem nem ao menos usar um simples `include` de PHP.

Usar `includes` do PHP (ou qualquer outro comando de qualquer outra linguagem que faça a mesma coisa), mesmo que seja algo amador, resolve apenas um dos problemas. O segundo problema era: como entregar isso para o cliente? Eu não poderia simplesmente enviar um pacote com vários arquivos `.php` para um cliente que trabalhava com ASP ou Python. Se ele me contratou para passar um PSD para HTML, preciso entregar para ele as telas do projeto em HTML/CSS/JavaScript estático.

Logo começamos a usar o `wget` para percorrer o projeto e transformar as páginas PHP em HTML estático. Não demorou muito para desistirmos disso. Embora o `wget` (ou até o `httrack`) faça quase tudo automaticamente, esse workflow não era o ideal. Precisávamos ter algo mais inteligente para executar esse processo. Foi aí que conheci algumas ferramentas para a geração de sites estáticos, como o Jekyll, Middleman e alguns outros.

Se você é bem novato ainda, vou explicar como se usa, mais o menos, o `include` do PHP logo a seguir. Mas se você já começou faz tempo nesse mundo e não conhece o Middleman nem o Jekyll, pule alguns parágrafos e leia mais sobre estes geradores de sites estáticos.

12.1 Para os novatos: Usando Includes PHP

O PHP é uma linguagem server-side e talvez a mais popular no mundo todo. Você não precisa ser programador para entender como funcionam os `includes`, mas **você precisa pelo menos aprender a instalar o Apache** para rodar o PHP na sua máquina.

`Include` significa `incluir` em português. É um comando do PHP (que também existe com outros nomes em outras linguagens server-side) que inclui um determinado arquivo dentro do outro. Coisa simples: imagine que você tem um código HTML assim:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8 
9 <header>
10   <h1 class="logo">Logo do site</h1>
```

```
11   <ul class="menu">
12     <li><a href="#">Home</a></li>
13     <li><a href="#">Contato</a></li>
14   </ul>
15 </header>
16
17 <p>Aqui vai o resto do site...</p>
18
19 </body>
20 </html>
```

O código do `header` é repetido em todas as páginas, como no exemplo que citei no início do capítulo. A ideia aqui é copiar esse código do `header` para um arquivo, que pode ser chamado de `header.php`, e o incluiremos em vez do código literal. Assim:

```
 1 <!DOCTYPE html>
 2 <html lang="en">
 3 <head>
 4   <meta charset="UTF-8">
 5   <title>Document</title>
 6 </head>
 7 <body>
 8
 9 <?php include "header.php" ?>
10
11 <p>Aqui vai o resto do site...</p>
12
13 </body>
14 </html>
```

Agora, toda vez que precisar mexer no código do cabeçalho, você só precisa modificar o código do arquivo `header.php` e todas as páginas estarão atualizadas.

Este é a técnica mais básica de todas e provavelmente você já até deve trabalhar dessa forma. Se este é o seu caso, vamos subir um nível agora.

12.2 Gerando sites estáticos com Middleman ou Jekyll

Aqui, você já precisa no mínimo entender como instalar uma GEM do Ruby.

Middleman e Jekyll são duas GEMS do Ruby que fazem a mesma coisa: criam sites com arquivos estáticos. O que vai decidir se você vai usar o Middleman ou o Jekyll é puramente gosto. Eu comecei

com o Jekyll, mas depois mudei para o Middleman, que se mostrou mais flexível. Mas tanto faz qual você vai usar, o que interessa são as vantagens que eles proporcionam para projetos de sites de pequeno e às vezes até de médio porte.

A ideia é que você produza um site estático, sem banco de dados algum, com arquivos totalmente modularizados, onde a lógica não está no server, mas nos próprios arquivos. Se você precisa fazer um site pequeno, institucional, sem muita firula, um site estático, com HTML, CSS e JavaScript é o suficiente. Não é necessário subir um Wordpress ou qualquer outro CMS complexo, que precisa de um servidor parrudo com banco de dados. Se é simples, mantenha simples.

Estrutura de diretórios

A estrutura de diretórios dos dois é bem parecida. Segue um exemplo de como seria o de um projeto em Jekyll.
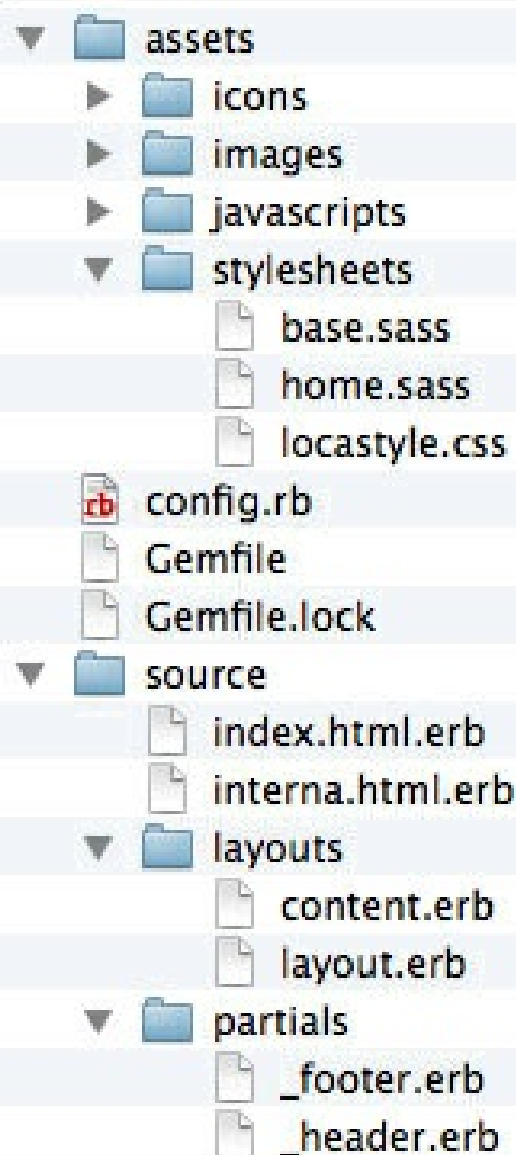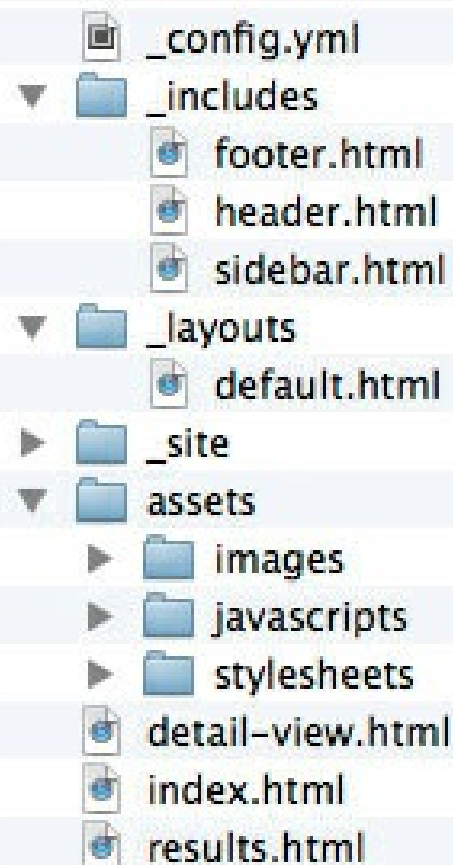
Fig. 12.1: Comparando a estrutura, o Jekyll parece ser mais simples.

A estrutura de pastas pode ser um pouco diferente entre os dois, mas eles têm muito em comum. Perceba que os dois possuem uma pasta chamada `layout`. Ela é onde guardamos os templates das páginas. Layouts da Home, Interna etc., são layouts com padrões diferentes, é ali que guardaremos esse código. Veja um exemplo do `default.html` do Jekyll:

```
1 {% include header.html %}
```

```
2
3 {{ content }}
4
5 {% include footer.html %}
```

Agora um exemplo do Middleman:

```
1 <%= partial 'partials/header'%>
2
3 <%= yield %>
4
5 <%= partial 'partials/footer'%>
```

Todos os geradores de arquivos estáticos se baseiam em um tipo de Template. O Template é como você vai manipular e usar as informações do seu site. O template padrão do Middleman é ERB, já o Jekyll usa uma marcação chamada Liquid (http://liquidmarkup.org), que é baseada em Ruby, mas que se parece mais linguagens de template, como o Mustache. O template Liquid é bastante usado em projetos com Shopify e Spree.

Ambos os geradores funcionam de forma muito similar. Sugiro que experimente os dois para tomar sua decisão final.

Sem banco de dados. Yeah!

A estrutura de código dos arquivos é muito simples de se entender, mas para alguns pode ser um pouco estranha por não ter familiaridade com estruturas de dados como YAML. Mas isso é simples e você aprende rápido, tenho certeza. Continue lendo para você ver como é fácil.

Para começar, você não mantém um banco de dados e é isso que faz toda a graça. O conteúdo do seu site fica guardado nos arquivos de cada página. Você não precisa levantar um servidor de MySQL. Todas as informações do site estarão nos arquivos que você criar para cada página. Ou seja, nada de queries, nada de templates tags do WordPress.

Um pouco sobre YAML

O formato YAML é conhecido pela facilidade de leitura. Ele foi criado para estruturarmos informação de maneira que seja fácil de entender e escrever. Ou seja, ele é um formato simples para escrevermos manualmente, mas que também possamos manipular via programação.

Qualquer arquivo que contém um bloco YAML – que geralmente é chamado de `front-matter` – será processado como um arquivo especial. O front-matter precisa ser a primeira coisa do arquivo e deve estar em um formato válido de YAML. Toda página do seu site, tanto no Jekyll quanto no Middleman, precisa começar com uma estrutura assim:

```
1 ---
```

```
2 title: Home
3 layout: default
4 ---
```

Simples? O bloco é demarcado pelos três traços no começo e no fim. **Tem** que ser três traços. Nem mais, nem menos. O código YAML são as duas variáveis `layout` e `title`.

Ambos os sistemas usam essa notação para organizar seus arquivos e informações. Você pode definir variáveis em cada um dos arquivos para fazer condições nas páginas. Se em uma página, por exemplo, eu quero que a sidebar apareça, a notação YAML dela será algo assim:

```
1 ---
2 title: Contato
3 layout: interna
4 sidebar: true
5 ---
```

A variável `sidebar` com o valor `false` foi criada por mim. No template, usarei algo parecido com isso, no Middleman:

```
1 <% if current_page.data.sidebar? %>
2   _A chamada do arquivo da sidebar vai aqui._
3 <% end %>
```

Lindo e simples. Quando buildar o projeto, o sistema já vai compilar os arquivos da maneira correta, com o código sidebar, no lugar que você pediu.

Aliás, para fazer o build é muito simples. No Middleman basta usar o comando `middleman build`. Ele vai compilar todo o projeto e colocar os arquivos prontos para o deploy dentro da pasta `build`. Já no Jekyll, basta rodar o comando `jekyll`. Ele vai jogar os arquivos prontos na pasta `_site`.

Agora, se o cliente quiser um painel administrativo para controlar o conteúdo do site, aí não tem jeito, aconselho a usar o Wordpress. Eu utilizo o Middleman/Jekyll para projetos em que o cliente pede para entregar os arquivos HTML estáticos. Como geralmente uso GIT, dou para ele o controle do repositório do projeto, contendo os arquivos de desenvolvimento, isto é, o código do Middleman/Jekyll, e também os arquivos do último build feito.

Esta foi só uma pequena introdução sobre os geradores de sites estáticos. Eu sugiro que você leia a documentação de cada um. Não é muito difícil de entender e você consegue fazer um teste rapidamente para aprender como funcionam.

- Documentação do Jekyll (http://jekyllrb.com)
- Marcação Liquid pelo Jekyll (http://liquidmarkup.org)
- Documentação do Middleman (http://middlemanapp.com)

- Listagem indicando vários outros geradores (https://www.staticgen.com)

Compartilhando o que sabe

Você já se perguntou por que tem alguns desenvolvedores mais populares que outros? Por que existem vários Zenos, Nandos e Shiotas por aí? Todos eles têm uma característica muito evidente: eles compartilham conhecimento. Eles ensinam algo para alguém todos os dias e fazem isso não apenas por que eles são solidários bonzinhos, mas principalmente porque isso aumenta o conhecimento deles. É um processo natural.

Um tweet do @fnando

"Acho que preciso escrever um artigo sobre flexbox... tem muita coisa que ainda fico viajando. :/"
https://twitter.com/fnando/status/566233046102597633

Parece óbvio, mas não é: quando você compartilha conhecimento, você aprende mais. Não sei como isso funciona e deve haver um monte de artigos científicos abordando o assunto, mas o fato é que, cada vez que você ensina alguém, inevitavelmente absorve mais conhecimento. Esse é um mecanismo natural e incrível que todo mundo possui. O processo de ensino faz você pensar mais e melhor. Ele o obriga a pesquisar profundamente sobre o assunto.

Cada vez que você escreve um artigo ou planeja uma palestra, você precisa resumir grandes assuntos em pequenas ideias. Você cria analogias para facilitar o entendimento da galera. Já tentou explicar o que é e para que serve a propriedade *float*? Tente explicar com detalhes para você mesmo, na frente do espelho, como funciona e qual a diferença dos valores *relative* e *absolute* da propriedade *position*; embora seja algo simples, você vai precisar pensar em uma explicação fácil de entender. É um exercício interessante e que abre horizontes que a maioria dos devs nem imagina.

Você não precisa se arriscar logo de cara a fazer palestras. Mas comece escrevendo um artigo. Comece publicando um pedaço de código no Gist ou em um JSFiddle qualquer. Não tenha medo do seu código ou da sua escrita. Tudo isso vai melhorar com o passar do tempo e você precisa começar de algum lugar. Você aprende mais levando bordoada dos críticos — e também dos Trolls — do que guardando seus textos e códigos apenas para você. Nada melhor do que ter alguém para apontar seus erros. Claro, se você tiver um amigo ou tiver amizade com outro desenvolvedor, muito melhor pedir a opinião dele antes de se jogar para os leões.

Mas meu apelo neste capítulo é simples: compartilhe o que você sabe. Simples assim. O Tableless nasceu exatamente dessa forma, assim como vários outros blogs. Diversos desenvolvedores são populares hoje porque meteram as caras em palestras nos eventos pelo país. Quer falar para poucas pessoas antes de enfrentar 100, 200, 300 ouvintes? Compareça em um encontro de desenvolvedores. Várias cidades têm Meetups onde você pode tratar a timidez falando poucas palavras para poucas pessoas. Apenas comece.

Textos extras

Estes são alguns textos publicados no Tableless, que talvez você não tenha lido e que são importantes para você que está começando. Tratam mais sobre comportamento e mercado do que de assuntos técnicos.

14.1 Último conselho: não queira ser o próximo Zeno

Quando montei este título eu sabia que ia ser polêmico e que haveria um burburinho nos corredores das interwebs. E sabe de uma coisa? Estou me lixando para que os outros falam por aí.

Sobre admitir que você é popular

Quando conversei com o Zeno horas depois de ter tido a ideia, expliquei os motivos para ele e comentamos sobre coisas que acontecem conosco por causa dessa “popularidade” na área. Eu realmente não quero que você seja o próximo Zeno Rocha. Não mesmo. O Zeno é o Zeno. Ele percorreu o caminho dele. Chegou longe e sei que vai chegar mais longe ainda. Mas uma coisa é certa: talvez você chegue mais longe sendo você mesmo do que querendo ser o Zeno.

Isso é difícil. Bastante. Como você diz que você é popular na área em que atua sem parecer arrogante, prepotente, mala ou um metido de nariz empinado? Não dá. A galera vai interpretar mal. Eu nunca soube fazer isso e não sei fazer até hoje. É por esse motivo que muitas vezes, no início da minha carreira, eu me escondia nos eventos, não usava os crachás com meu nome e essas coisas. Leva tempo para assimilar. Mas admitir que você é popular (http://bit.ly/1stoXjx) tem mais a ver com as atitudes que você vai tomar dali para frente do que se importar com o que os outros vão falar. É muito sobre como você vai lidar com a situação de ser popular e como você vai usar essa vantagem (eu seria imbecil de dizer que isso não é uma vantagem) para melhorar sua carreira e, claro, ajudar outros devs.

E sabe de uma coisa? Tem que ter coragem para bater no peito e admitir para você e para os outros que você é uma estrelinha na área. Não é questão de ser humilde. É questão de entender qual o seu papel na vida dos outros, na sua carreira e principalmente na sua própria vida.

Cansei de receber e-mails de devs iniciantes dizendo que queriam ser igual a mim. Percebi logo que isso não era saudável para o cara do outro lado e muito menos para mim. Sim, você precisa seguir os passos de alguém. Isso é algo inteligente de se fazer. É assim que a humanidade evolui. Mas entenda: seguir os passos de alguém não quer dizer que você tenha que ser a outra pessoa. O Zeno foi ele mesmo durante todo o caminho. Eu fui eu mesmo durante todo o meu caminho e você precisa ser você mesmo, seguindo o seu caminho. Se não, você vai falhar.

Quando eu tinha minha empresa, contratei um consultor de negócios para nos guiar. No início ele quis entender qual o papel da empresa e dos sócios no mercado, como atuávamos e qual a nossa influência na área. Quando ele soube que eu era um cara popular no meio, ele disse assim: “O que você vai fazer quando não for mais o garoto do momento?”. Eu admito que nunca havia pensado naquilo e isso ficou martelando durante alguns dias na cabeça. Isso me fez pensar bastante sobre

várias coisas, principalmente em como eu estava usando ou subutilizando a minha imagem no mercado.

Quando eu (ele) me convenci de que um dia esse dia chegaria, fiquei mais tranquilo e entendi que esse é o fluxo natural e saudável. A popularidade vem e vai. Se você não está preparado pra isso, seu ego explode e o leva junto. Um dia vão parar de falar o seu nome para falar o nome de outra pessoa. Foi o que aconteceu comigo, várias vezes. Não só o Zeno, mas outros caras apareceram e vão continuar aparecendo. É importante que isso aconteça. Você é forçado a reavaliar os seus erros e principalmente a avaliar o que os outros fizeram e o que você não fez para chegar lá. Mas o segredo é fazer essa avaliação **agora**, sendo ou não popular.

Sobre mudar o Mundo

Outra coisa difícil de entender é que nem todo mundo nasceu pra mudar o mundo (bit.ly/sobre-mudar-mundo). Talvez você não fique popular. Talvez você não seja reconhecido pelas pessoas que você ajudou. Talvez você fique chateado e descubra que web não é a sua praia, embora você queira muito que seja. Se isso acontecer, esse processo pode ser feito em outro mercado, em outra área, com outras pessoas. Você só precisa destes três componentes básicos e estará a salvo: vontade, tempo e curiosidade. Estes seriam os componentes importantes para qualquer tarefa que precise ser executada.

Quando resolvi fazer um artigo com o título *Não seja o próximo Zeno Rocha*, um monte de gente interpretou errado. Mas você que é inteligente o bastante vai perceber que o que eu quero mostrar nesse artigo é que qualquer um pode chegar lá sem ser o Zeno, o Maujor, o Eduardo Shiota, o Nando Vieira, o Diego Eis, Leo Balter, Lucas Mazza, Jean Carlo Emer, Jaydson Gomes, Bernard de Luna, Daniel Filho, Almir Filho, Caio Gondim ou qualquer outro desenvolvedor conhecido na área, mas sendo você mesmo. Com as suas características e habilidades.

Se eu tivesse que adiantar apenas uma dica, ela seria essa: ajude e ensine outros devs.

Muito provavelmente eu não deveria ter publicado este artigo. Mas já que foi, espero que não o interprete errado. A dificuldade de falar desse assunto é inversamente proporcional à facilidade dos trolls de serem imbecis.

14.2 Sobre os Pavões do Front-end

Eu gostava quando o mercado de front-end era mais simples. Quando não havia tantos eventos. Quando não havia tanta gente querendo ser a estrela da vez. Eu gostava quando o mercado era pequeno, onde a maioria das conversas rolava em listas de discussões. Quando quase não havia blogs sobre o assunto e os blogueiros que mais falavam tecnologia e também front-end eram profissionais de outras áreas. Eu gostava desse tempo.

Não quero parecer um velho chato ranzinza falando. Mas é que há uma linha muito sutil entre querer

se promover e querer que o mercado cresça de forma saudável. Hoje em dia tem tanta gente querendo ser a referência da área que seria trágico se não fosse tão engraçado. Todos eles dizem que estão mudando o mercado. Que você precisa se levantar e fazer alguma coisa. Que você precisa ser ousado e todo o blá blá blá de que você já deve ter ouvido falar milhões de vezes por aí. Muitos deles até estão se esforçando para que esse discurso seja real. Em alguns, você até consegue ver que estão realmente preocupados com o rumo do mercado. Em outros, fica na cara que eles querem ter seus minutos (ou anos) de fama.

Eu recebo artigos de voluntários todos os dias pelo Tableless. São pessoas que até então estavam anônimas. Todas elas expressam um desejo de compartilhar conhecimento. Muitos deles nunca palestraram, muitos deles nunca nem sequer escreveram um artigo. Mas eles querem ajudar de alguma forma a comunidade a crescer. Eles querem ter voz também. Querem falar para os outros, que como eles, sabem que não precisam ser um showman para ajudar, basta ter vontade.

Não sou contra quem faz self-promotion. Pelo contrário, você precisa de uma pitada de self-promotion para alavancar sua carreira, seu blog, seu nome etc. Se autopromover é algo que você precisa usar, mas usar bem. Uma coisa é se autopromover para se ajudar, outra é se autopromover para ajudar um grupo, um propósito. Também não sou contra alguém querer ter seus minutos de fama. Isso é normal e às vezes é o pagamento para aquele trabalho gratuito que o cara tem feito para a comunidade. O problema é quando a fama se torna a recompensa final.

É por isso que gosto dos tempos antigos. O mercado chegou até aqui por causa de pessoas que dedicavam seu tempo, de graça, para manter fóruns, listas de discussão, encontros semanais, tutoriais etc. por puro amor à profissão.

14.3 Sobre o design oco

Quando construímos um padrão de interface para qualquer tipo de produto digital, é necessário que esse padrão se perpetue, caso contrário, seu trabalho não valerá de nada. Esse padrão precisa ser protegido durante todo a produção do projeto, offline ou online, não importa. Para que essa perpetuidade seja real, você e sua equipe precisarão defender esse padrão e, para defendê-lo, você precisa de argumentos. É aí que o problema aparece.

Não sei se você já tentou convencer alguém de que o design e o trabalho de *user experience* são importantes. Eu já tentei algumas vezes e nenhuma foi agradável. O trabalho de design e da experiência do usuário são coisas intangíveis. Eu não consigo mostrar o quanto a mudança de design de um site aumentarão as vendas, antes de esse design ser publicado.

Manter um design antigo só porque vende pode ser a decisão mais racional, mas está longe de ser a ideal. Era assim com o Google até eles decidirem mudar toda sua interface e a experiência de uso dos seus produtos. Um concorrente direto do Google é a Apple, uma empresa que respira design. O Google era uma empresa essencialmente dominada por programadores. O Google soube em algum momento que a falta de design — ou o design ruim — atrapalharia seus planos em longo prazo. Sim, o design é um fator decisivo para diversos usuários. Mas é comum as empresas não priorizarem

decisões de mudanças no site/produto baseadas nas necessidades de design ou UX.

É engraçado pois todo mundo gosta de ver algo bonito. Você compra coisas físicas julgando principalmente dois aspectos: design e facilidade de uso. Você não compra algo porque é feio e difícil de usar. Ou compra?

Com software isso não deveria ser diferente. E na verdade não é. O usuário geralmente prefere algo atraente, que funciona bem e que não o faça perder tempo usando.

Arranjando argumentos

Mas como nós arranjamos argumentos para defender um padrão visual? Como arranjamos argumentos para defender um novo design para produtos ou sites que já estão dando certo?

Simples: não faça aquele design que é bonito mas ordinário. Manja aquele design bonito, mas oco? É esse que você não pode fazer.

Você encontra esse tipo de design em toda parte. Dê um pulo no Theme Forest ou até no Dribbble (https://dribbble.com/). Já rolaram algumas discussões sobre a superficialidade (http://blog.intercom.io/the-dribbblisation-of-design/) dos designs que rondam o Dribbble e outros serviços parecidos.

Nenhum elemento vem do além

Eu quero dar um exemplo claro usando ainda o Material Design do Google. Se você ainda não viu a documentação deles, vamos ver agora a parte onde eles explicam sobre botões, neste link (http://www.google.com/design/spec/components/buttons.html).

No Material Design existem três tipos de botões: o *floating button*, *raised button* e o *flat button*. No início, parece que não é necessário ter tantos tipos de botões, mas se você pensar direito, há diversos cenários diferentes no ambiente mobile e também no desktop. Cada cenário tem um contexto diferente e que exige uma atenção maior ou menor para as ações.
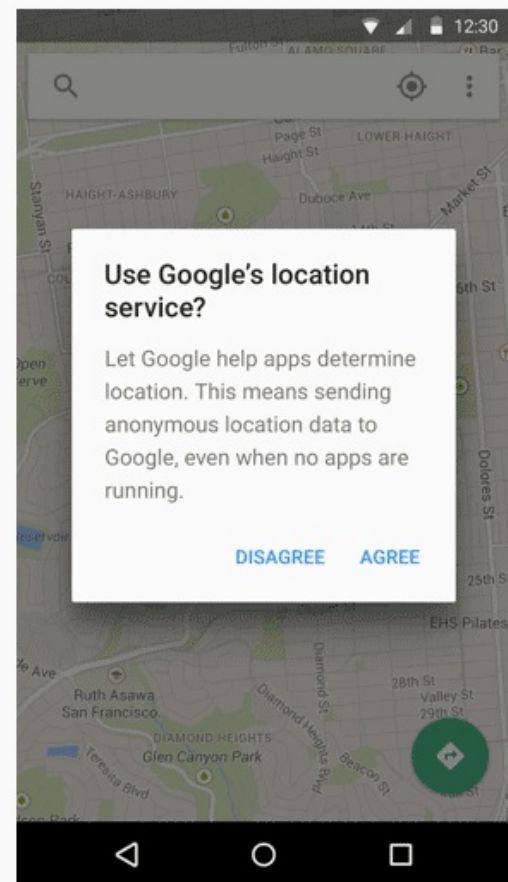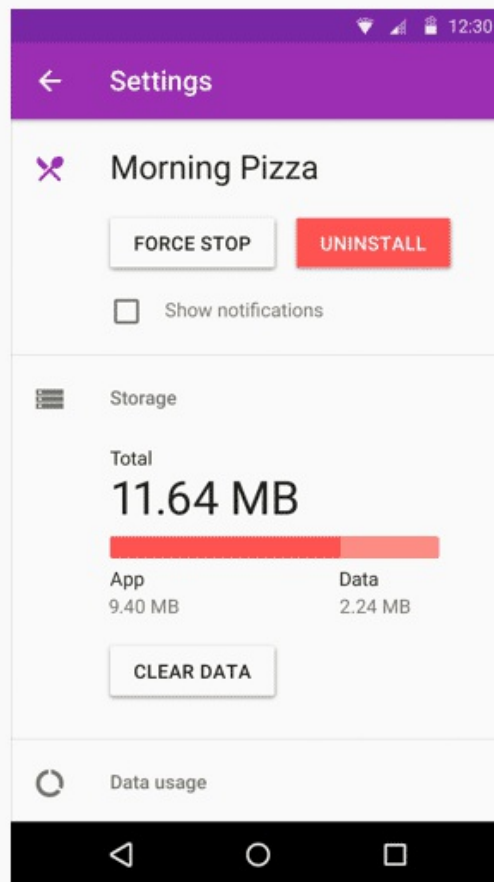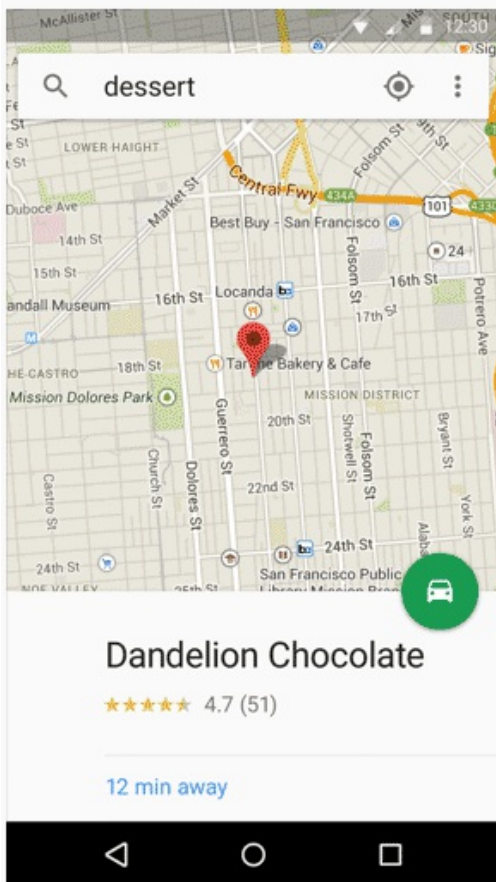
Fig. 14.1: Exemplos de elementos que o Google usa no Material Design.

Eles também tomaram cuidado para indicar a frequência de uso de cada tipo de botão, além dos lugares específicos onde cada tipo de botão é usado. Entenda que há ciência nisso aí! Se há um padrão, há organização. Se há organização, o cérebro do usuário entende melhor as regras de uso, as restrições etc.

A quantidade de elementos de um mesmo tipo também não será problema, já que há restrições de uso. Claro, você não vai adotar uma coleção de 10 botões. Nesse caso, inteligência e bom senso precisam ser uma qualidade na sua equipe.
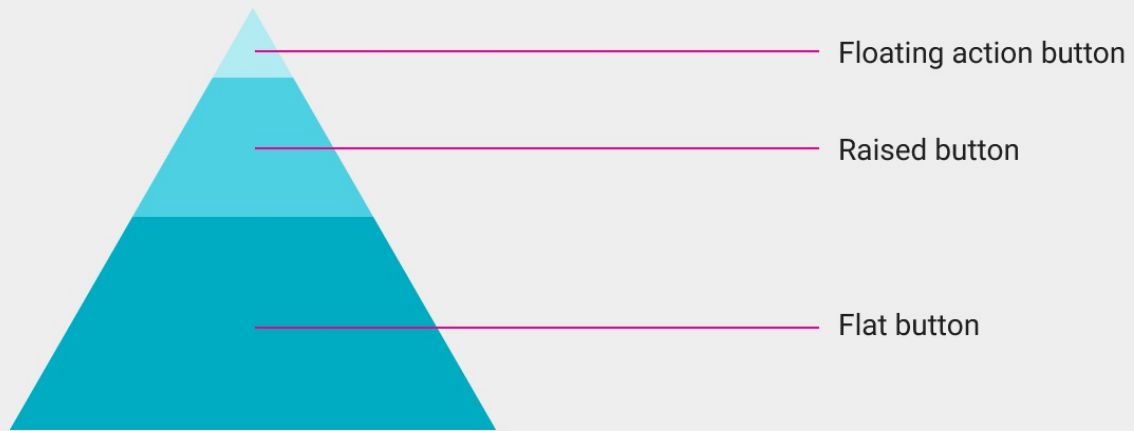
Fig. 14.2: Essa restrição de prioridade permite que o usuário saiba quais tipos de botões são mais importantes que outros.

Quando há um motivo para cada elemento na tela, você ganha argumentos. Se você simplesmente coloca um elemento na tela porque achou que era tendência do mercado, você perde. Se você coloca porque achou bonitinho, você perde. Esse é o design oco, vazio, ordinário.

Você também ganha argumentos descobrindo as falhas e as vantagens do site/produto atual. Tendo informações sobre o projeto atual, você não precisa sair da estaca zero. Você saberá exatamente como as vendas acontecem ou como elas são perdidas. Isso pode ser levado para a próxima versão, pode ser testado e aprimorado.

Esse é o ponto onde é preferível realinhar seu visual e os comportamentos de interações do que fazer um redesign total (http://alistapart.com/article/redesignrealign). Quando você realinha, você evolui o design. Você filtra as coisas ruins. Quando há um redesenho total do site, um mundo de incertezas vem à tona novamente e, claro, seus argumentos terão que ser bem fortes para defender um redesign completo. Quando você realinha, você refaz uma pequena parte do website/produto. Se você faz um realinhamento de vários componentes durante o ano, ao final você terá um produto novo, testado de forma modular e sabendo que cada componente tem um sentido para estar ali.

Obviamente você só pode fazer um realinhamento se tiver uma boa base estrutural. Há coisas que só se resolvem refazendo tudo. Outras não. O redesign nesse caso pode ser benéfico por agir como um reboot, possibilitando que realinhamentos sejam possam ser feitos nas próximas vezes.

Enfim

Na verdade, esse é um pensamento ainda pouco explorado em um monte de empresas. Na Locaweb estivemos durante um bom tempo no processo de reboot. Infelizmente, é um processo doloroso. Mas há a certeza de que os próximos passos serão mais fáceis. Os argumentos serão mais fortes e teremos números importantes falando ao nosso favor. Aqui é um pouco mais fácil porque temos um framework que nos ajuda no processo de padronização de (quase) todas as interfaces da empresa. Mas esse é outro papo.

Até mais e obrigado pelos peixes

Eu sempre gostei de estar perto dos novatos da área. Eu fui novato um dia e sei a dificuldade que é arranjar informações durante o processo de aprendizagem. O intuito deste livro foi simplesmente apresentar para você as tecnologias e metodologias que são ou estão populares no mercado no momento. Espero que você tenha gostado bastante e que as informações sirvam de referência durante muito tempo.

O mercado de front-end brasileiro tem crescido bastante. Isso é bom. Isso é maravilhoso. Isso é pica das galáxias. Mas também é algo ruim. Como é muito fácil aprender HTML e CSS, ser dev front-end acaba sendo uma porta bastante acessível para muitas pessoas. Nem sempre essas pessoas se tornam profissionais engajados. Eles querem simplesmente ganhar uma grana, sobreviver e pronto. Para eles, essa profissão é como qualquer outra. Não estou dizendo que eles estão errados. Isso acontece em qualquer profissão, em qualquer país. Para esses, tanto faz o que falamos durante todo livro. Tanto faz os eventos, os meetups, as reuniões que a galera da comunidade faz durante o ano. Eles devem ser deixados de lado... Este livro foi criado para outro perfil de profissional.

Ficarei muito grato se você compartilhar comigo seu ponto de vista sobre este livro. Tentei fazer algo bom aqui e tenho certeza que falhei em várias partes. Talvez em alguns momentos eu poderia ter me aprofundado mais, em outras eu poderia ter falado de menos. Por isso, fale para mim a sua opinião, os pontos que eu poderia ter melhorado ou se há algo que eu esqueci de incluir. Fale comigo pelo e-mail livro@tableless.com.br.

15.1 Devs que você deve seguir

Essa coisa de criar listas dos *melhores* front-ends não é legal e geralmente não é bem recebida por várias pessoas. Por esse motivo eu fiquei com um pé atrás, mas criei uma lista de profissionais que são referências para mim.

Entenda que você precisa ter referências e opiniões de várias pessoas do mercado. Hoje eu fico feliz por ter vários nomes na cabeça para indicar. Quando comecei nessa área, não havia muitos desenvolvedores com disposição para ajudar ou que publicavam artigos técnicos. Tive que aprender na marra, encontrando conteúdo quase que exclusivamente gringos. Hoje o mercado nacional é muito, mas muito próspero. Por isso, aqui vão alguns nomes para você seguir:

Almir Filho trabalha na Globo.com desde sempre. É um cara descolado, alegre e bastante aberto para conversas de qualquer tipo. Você já deve ter ouvido falar sobre ele por causa do blog Loop Infinito, que ele mantém junto com o Caio Gondim (http://loopinfinito.com.br/) e também agora ele faz parte do Zofe, um podcast exclusivamente para assuntos sobre desenvolvimento front-end.

- Twitter: @almirfilho
- Blog: http://loopinfinito.com.br/
- Palestras: https://speakerdeck.com/almirfilho

Davi Ferreira é um desenvolvedor bem tímido na comunidade, mas um dos mais competentes que

conheço. Publica alguns artigos no Tableless e principalmente no Medium. Suas dicas no Twitter e seus artigos técnicos são minhas referências.

- Twitter: @davitferreira
- Blog / Site: http://www.daviferreira.com/posts
- Palestras: http://www.daviferreira.com/page/decks
- Perfil no Tableless: http://tableless.com.br/author/davitferreira/

Eduardo Shiota é o japa mais legal da comunidade. Super atencioso e sempre disposto em ajudar qualquer um sobre JS, CSS, HTML etc. Ama gatos e não vai hesitar em dar uma opinião sincera sobre qualquer assunto, principalmente sobre quais decisões tomar em sua carreira.

- Twitter: @shiota
- Blog: http://eshiota.com/
- Palestras: https://speakerdeck.com/eshiota

Jean Carlo Emer surgiu do nada. Seu jeito tranquilo e cauteloso que ele expressa suas opiniões é uma marca registrada. Mesmo assim suas opiniões são sempre fortes e diretas. Além de fazer uma pancada de palestras de ótima qualidade pelo Brasil, ele ajuda a movimentar a comunidade de front-end em Porto Alegre.

- Twitter: @jcemer
- Blog / Site pessoal: http://jcemer.com/
- Palestras: http://jcemer.com/talks.html
- Perfil no Tableless: http://tableless.com.br/author/jeancarloemer/

Leo Balter é a cara da humildade. Ele não hesitou em mencionar detalhes pessoais em sua trajetória para a Bocoup, que fica em Boston, a fim de matar a curiosidade de outros devs sobre como é o processo de ir trabalhar fora do Brasil. É amigo de todos e super solícito com qualquer um que precise de ajuda sobre JS. Para você ter ideia, ele é commiter do time do QUnit do jQuery. Ele respira JavaScript.

- Twitter: @leobalter
- Blog: http://leobalter.github.io/
- Palestras: http://pt.slideshare.net/leobalter

Nando Vieira é back-end, mas o cara tem a pegada no front-end. Como um bom full stack, ele conhece bem os dois lados da moeda. Ele tem opiniões bem ácidas sobre tudo que envolve desenvolvimento web. Até por isso ele é odiado por uns e amado por outros. Talvez seja inveja da oposição. Eu sou do time que ama o cara. <3

- Twitter: @fnando
- Blog: http://simplesideias.com.br/
- Palestras: https://speakerdeck.com/fnando

Rafael Rinaldi, para mim, é um sinal de esperança! O mercado precisa de novas caras palestrando,

escrevendo, gravando vídeos e com ideias e opiniões diferentes dos devs mais antigos. O Rinaldi é o sinal de que isso está acontecendo. Ele apareceu do nada e tem feito algumas palestras bem interessantes pelos eventos Brasil afora.

- Twitter: @rafaelrinaldi
- Palestras: https://speakerdeck.com/rafaelrinaldi
- Site: http://rinaldi.io/

Sérgio Lopes toma muito redbull, certeza! Ninguém fica triste assistindo suas palestras. Em seus artigos ele sempre aborda assuntos novos, que estarão na boca do povo muito tempo depois. É o cara que ajudou a popularizar o Responsive Web Design aqui no Brasil. Gente muito fina, agora mais do que nunca! ;-)

- Twitter: @sergio_caelum
- Blog/Site: http://sergiolopes.org/
- Palestras: http://sergiolopes.org/palestras/

Existe uma série de outros devs que eu deixei fora da lista, mas se eu fosse listar todos daria outro livro (uia, boa ideia). Mas só citando alguns: Clauber Stipkovic, Reinaldo Ferraz, Raphael Fabeni, Bernard de Luna, Giovanni Keppelen, Daniel Filho, Caio Gondim, Clécio Bachini, Zeno Rocha, Jaydson Gomes, Paulo Ragonha, Gabriel Zigolis, Mauricio Soares e vários outros!

E os gringos?

Não vou fazer uma descrição bonita de todos os gringos, por isso vou colocar apenas o Twitter deles aqui.

- @timberners_lee (nem preciso falar, né?)
- @zeldman (para mim, esse cara é o Deus da web)
- @meyerweb (ajudou a popularizar o CSS)
- @chriscoyier (criador do CSS Tricks)
- @glazou (esse cara faz parte do CSS Working Group do W3C)
- @davidwalshblog (dev senior na Mozilla)
- @addyosmani (dev no Google)
- @mdo (cocriador do Bootstrap)
- @fat (cocriador do Bootstrap)
- @beep (dono do termo Responsive Web Design)
- @lukew (sobre mobile web)
- @bradfrost (sobre mobile web)
- @daveshea (Mezzoblue)
- @maxvoltar (designer no Dropbox)
- @jasonsantamaria (designer top de linha)
- @simplebits (cofundador do Dribbble)
- @paul_irish (js man)

15.2 50 palavras

Com a ajuda da galera do Twitter (Valeu, pessoal!), consegui juntar 50 palavras (um pouco mais, na verdade) que você pode ouvir muito por aí. São palavras relacionadas a tecnologias, metodologias, editores, linguagens de programação, frameworks e outras coisas que envolvem o mundo do desenvolvimento web de alguma forma. Vou comentar algumas delas, mas você pode ver a lista completa ao final.

Web Components

Mesmo que tenha dado uma caída no final de 2014, Web Components tem sido um assunto muito recorrente em palestras e em vários blogs técnicos. Muitos defendem a ideia, outros ainda acham que precisa ser melhorado, mas todo mundo entende que é um passo para um caminho em direção ao desenvolvimento componentizado, de verdade. Eu ainda não fiz nenhum projeto utilizando Web Components e não vou usar enquanto eu não precisar e enquanto não tiver algum tipo de suporte nativo nos browsers.

ES6

Essa nova versão do JavaScript tem sido bastante comentada porque ela traz uma série de mudanças relevantes para a linguagem. O JavaScript é uma linguagem que se popularizou muito durante os últimos 5 ou 6 anos. Várias novas tecnologias têm sido baseadas no JavaScript, por exemplo, uma grande parte do ferramentário que temos hoje, sem falar no NodeJS. A versão 6 do EcmaScript vem para resolver alguns gaps que a linguagem tinha e também trouxe uma série de outras coisas interessantes, herdadas de frameworks populares como CoffeeScript, além de ajudar os frameworks e bibliotecas existentes a se tornarem mais inteligentes e performáticas.

SPA, Data binding, Angular, React e companhia

Mais do que ES6, Web Components ou qualquer outra coisa, eu ouvi falar mais foi sobre SPA, Angular, React e companhia. Toda essa tecnologia criada em volta dos SPAs foi muito boa para comunidade e nos ensinou um bocado sobre performance e otimização. Eu não via um balanço tão grande sobre o assunto desde a utilização de Ajax pelo pessoal do Gmail. A parte ruim são os manés querendo fazer qualquer sitezinho simples de conteúdo usando algo como Angular em vez de usar iniciativas como o SennaJS, criado pelo Eduardo Lundgreen junto com outros devs. O SennaJS é um engine para a criação de SPAs de forma inteligente, que pode ser usado em projetos de todos os tamanhos.

As 50 palavras

Segue a lista. Aproveite. Essa lista foi montada pela galera neste Gist (bit.ly/frontendwords2014).

1. Web Components
2. Tooling
3. ES6
4. LeanUX
5. Data Binding
6. Mobile First
7. Performance
8. Material Design
9. SASS
10. Sublime Text
11. Polymer
12. Grunt
13. GulpJS
14. Arquitetura CSS
15. NodeJS
16. Atomic Design
17. Bower
18. Yeoman
19. SVG
20. Responsive Web Design
21. Flexbox
22. CSS Grid
23. Firebase
24. Continous Integration
25. Wearables
26. Design Thinking
27. Testes A/B
28. io.js
29. IE11
30. Jekyll
31. Middleman
32. FirefoxOS
33. Brakets
34. WebAPIS
35. Docker
36. React
37. Shadow Dom
38. Build-first
39. Vagrant
40. Acessibilidade
41. Reactive JavaScript
42. Functional JavaScript
43. Ansible
44. VIM
45. Shell

46. MV*
47. Inspector / Developer Tools
48. WebKit
49. W3C DRM
50. WebRTC
51. Mobile debugging
52. Fault Tolerant